HelixCore

Helix Core Server Administrator Guide

2020.2

November 2020

PERFORCE

www.perforce.com

Contents

How to use this guide

Installation, configuration, and management of Helix server (`p4d`) by a:

- System administrator - install and configure, ensure uptime and data integrity
- Helix server administrator - users, depot access, and authentication

This section provides information on typographical conventions, feedback options, and additional documentation.

Syntax

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

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
`-a -b`	Both *a* and *b* are required.
`{-a | -b}`	Either *a* or *b* is required. Omit the curly braces when you compose the command.
`[-a -b]`	Any combination of the enclosed elements is optional. None is also optional. Omit the brackets when you compose the command.
`[-a | -b]`	Any one of the enclosed elements is optional. None is also optional. Omit the brackets when you compose the command.
`...`	Previous argument can be repeated. ■ `p4 [g-opts] streamlog [ -l -L -t -m max ] stream1 ...` means `1` or more stream arguments separated by a space ■ See also the use on `...` in Command alias syntax in the *Helix Core P4 Command Reference* **Tip** `...` has a different meaning for directories. See Wildcards in the *Helix Core P4 Command Reference*.

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other documentation

See https://www.perforce.com/support/self-service-resources/documentation.

Tip
You can also search for Support articles in the Perforce Knowledgebase.

Earlier versions of this guide

- 2020.1
- 2019.2
- 2019.1
- 2018.2
- 2018.1
- 2017.2
- 2017.1
- 2016.2
- 2016.1
- 2015.2
- 2015.1
- 2014.2
- 2014.1

Note
The "Deployment architecture" on page 369 chapter used to be a separate book named "Helix Versioning Engine Administrator Guide: Multi-site Deployment": 2019.1, 2018.2, 2018.1, 2017.2, 2017.1, 2016.2, 2016.1, 2015.2, 2015.1, 2014.2, 2014.1

What’s new in this guide

This section provides a summary with links to topics in this Helix Core Server Administrator Guide. For a complete list of what's new in this release, see the Release Notes.

2021.1 release

- You can enable the display of real-time monitoring values. See `--show-realtime` under "General options" on page 501 in "Helix Core server (p4d) Reference" on page 499.
- You can verify a subset of database tables. This is faster than verifying all of the database tables. See `-xv` under "Server options" on page 500 in "General options" on page 501 in "Helix Core server (p4d) Reference" on page 499.
- "SSL and TLS Protocol Versions" on page 127 have configurables on the client-side.

> **Note**
> If you are licensed for Helix Core version 2021.1 patch 1 or greater, Helix4Git licensing is included at no extra charge.

2020.2 release

- During upgrades to a new server version, the upgrade steps now execute in the background (applies to 2019.2 and later upgrade steps). This can improve server availability and replication performance during upgrades.
  - While you are upgrading, you might want to get the status of the upgrade steps. Consider using the the new p4 upgrades command, which is documented in the *Helix Core P4 Command Reference*.
- Any new file being shelved that has the same content as an existing shelved file now refers to the existing archive file instead of creating a duplicate archive file.
  - To avoid overwriting the content of shelves that share archives, the archives of the new shelved files now have an additional numerical suffix. For example, 1.1.1.gz instead of 1.1.gz.
- The "Helix Proxy" on page 482 service can now be configured to use a different path for databases by either setting the P4PROOT environment variable or by using the `-R` proxy option. By default P4PROOT is the same as P4PCACHE, and both databases and archives will reside on the same path.
- Enhancement of "Background archive transfer for edge server submits" on page 453: if the administrator has not enabled background submit, the `-b` option of p4 submit is ignored and standard submit behavior occurs.

- "Failover" on page 191 from a mandatory standby server when the master is not participating used to require specifying `-s <serverID>`. Now, failover for this scenario includes checking the `ReplicatingFrom` field of the standby server spec for the master's serverID when `-s` is not specified on the command line.
  - Part of the failover process involves stopping the journalcopy and pull threads. If the failover process fails, those threads needed to be restarted manually. Now any `pull -L`, `pull -u`, or journalcopy threads that were configured using `startup.N` configurables will automatically be restarted if the failover process did not succeed.
- For your end-users of P4V 20.3 and later, you can set the maximum number of files to load into the Depot tree at any one time. See `P4V.Performance.DirFetchSize` in "Performance-related P4V properties" on page 94.

2020.1 release

- Support for "Helix Authentication Service" on page 134.
- The storage upgrade process is now visible through the `p4 monitor` command. See p4 monitor in *Helix Core P4 Command Reference*.
- A heartbeat-related trigger or extension can be part of a solution to monitor whether a server is responsive. See "Triggering on heartbeat (server responsiveness)" on page 349
- TLS 1.3 is now supported, but TLS 1.2 remains the default. See ssl.tls.version.max in the *Helix Core P4 Command Reference*.
- A unique command identifier is now visible through structured logging for schema version `50`. See "Logging" on page 206 and the Support Knowledgebase article on Structured Server Logs.
- Global labels can now be updated from edge servers using either `p4 tag -g` or `p4 labelsync -g`. See p4 tag and p4 labelsync in *Helix Core P4 Command Reference*.
- The host field in the protections table now allows multiple IP addresses or CIDR matchers to be specified on a single line with a comma-separated list. See p4 protect in *Helix Core P4 Command Reference*.
- The sequence numbers used in `db.protect` are no longer contiguous. This allows new lines to be added without rewriting the whole table.
- Standard users can now view the storage record table.
- A new configurable has been added to suppress the generation of digests during a storage upgrade. See lbr.storage.skipkeyed in *Helix Core P4 Command Reference*.
- The scan of the protections table at command start is now lockless. This means that updates to the protections table on busy servers will experience less lock contention.

Documentation-related

- Helix Core Server Administrator Guide is now a single volume instead of being split between "Fundamentals" and "Multi-site Deployment". What was previously "Helix Core Server Administrator Guide: Multi-Site Deployment" is included here under "Deployment architecture" on page 369.
- For clarity, "multi-server environment" means what was formerly called "distributed environment" and "distributed" is primarily associated with *Using Helix Core Server for Distributed Versioning* (DVCS).
- This manual now explains how to get the latest patch release. See "Patching the server" on page 70.

2019.2 release

Upgrading

The 2019.2 upgrade steps are significantly different from any prior release. See Upgrading the server.

Improvements to structured logging

Structured logging has a new format so it can be more helpful for the analysis of performance. See Logging and structured log files.

failed-over trigger

A new type of trigger, **`failed-over`**, can run when a standby server becomes the new master. See "Triggering on failed-over" on page 351.

2019.1 release

- Carefully follow the steps in the chapter on Upgrading the server because they are different from those of upgrades to versions prior to 2019.1.
- **Extensions**, a new technology that is similar to Triggers, but with advantages and additional capabilities. See "Triggers and Extensions" on page 284.
- When the server is busy with the maximum number of commands and therefore blocking **`standard`** users,the **`super`** and **`operator`** "User types" on page 231 can still issue a subset of commands. See "Limiting simultaneous connections" on page 265 > "Too many commands" on page 265.

- You can save disk space when creating an archive depot by using the option that includes lazy copies, which are small references to the location of potentially large files. A new database table, `db.storage`, replaces the `db.archmap` table to provide a link count for archive files on the server. This tracking reduces the complexity of identifying lazy-copies, allowing +S*n* files to be lazy copied by reference instead of being duplicated with their full contents. See `p4 archive -z` in*Helix Core P4 Command Reference*.
- Faster verification of archives (depot files) with `p4 verify`
  - such verification can also be done with a new command, p4 storage `-v`
- Display, verify, or update physical archive storage with the new command, p4 storage
- Faster p4 obliterate
- The procedure for setting up a high availability server for "Failover" on page 191 has changed. See "A high availability standby within an existing installation should not be initially deployed as mandatory." on page 193
- Note that *Helix Core P4 Command Reference* indicates that the net.autotune configurable is on by default.
- (Doc-only change: The "How protections are implemented" on page 162 topic has been expanded.)
- End-users can benefit from the "Background archive transfer for edge server submits" on page 453.
- "Edge-to-edge chaining" on page 456: an edge server can be configured to connect to another edge server without needing to sync from a remote commit server. See also "Commit-edge" in "Deployment architecture" on page 369 and "Filtering metadata during replication or edge-to-edge chaining" on page 403

2018.2 release

- "Failover" on page 191 to a new master server is now an easier process
- Installation support for SUSE Linux Enterprise Server - see "Linux package-based installation" on page 35
- Clarification on when trigger-based authentication can fall back to a password request: "Single sign-on and auth-check-sso triggers" on page 333
- If you want to write a trigger that requires users to log in with additional security, see "Triggering for multi-factor authentication (MFA)" on page 340
- "Multi-factor authentication" on page 146 (MFA) is the current name for a feature that was originally introduced as second-factor authentication (2fa)
- Helix SAML is a new feature for authentication

2018.1 patch

If you want to write a trigger that requires users to log in with additional security, see "Triggering for multi-factor authentication (MFA)" on page 340

Installation support for SUSE Linux Enterprise Server 11 and 12 - see "Linux package-based installation" on page 35

2018.1 release

You no longer need to use the `-z` option to restore a compressed checkpoint or journal. This allows the chaining of files for the restore. For example:

```
p4d -r . -jr checkpoint.42.gz journal.42 journal.43 journal
```

See the topic named ""Database corruption, versioned files unaffected" on page 186", which has a Note about Version 2018.1

See graph-push-reference triggers at "Triggering with depots of type graph" on page 345

A new structured log, `ldapsync.csv`, has been added to record the activity of p4 ldapsync. See "Enable and configure structured logging" on page 208.

2017.2 release

"Triggers for external file transfer" on page 347

See "Triggers for external file transfer" on page 347

Server background tasks

See p4 bgtask in the Command Reference

Parallel threads

p4 shelve now accepts the `--parallel` flag to specify that multiple files should be transferred in parallel, using independent network connections from automatically-invoked child processes. In addition, new configurables `net.parallel.shelve.*` allow p4 shelve to automatically use parallel threads to transfer files. Please see `p4 help shelve` and `p4 help configurables` for complete information.

The net.parallel.sync.svrthreads configurable reduces the number of parallel transmit threads used by sync commands when the total number of "user-transmit" threads (from all commands) running concurrently in the server would exceed the value of this configurable. Server monitoring must be enabled for this new configurable to take effect.

Complete replication for graph depot archives

Edge servers support syncing file content from graph depots. Replication supports graph depots that contain pack files, loose files, or a mixture of the pack files and loose files.

New content can be pushed by using the Git Connector or committed with p4 submit or p4 merge.

For information about depots of type graph, see:

- Working with depots of type graph in the *Helix Core P4 Command Reference*.
- Overview in the *Helix4Git Administrator Guide*.

Overview

Read *Solutions Overview: Helix Version Control System* before you read this guide.

Basic architecture

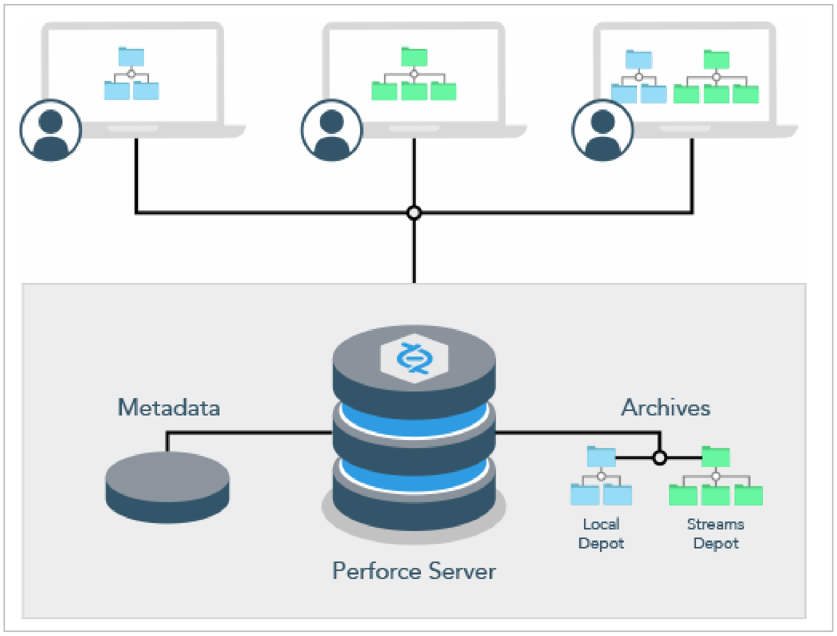
The simplest Helix server configuration consists of a client application and server application communicating over a TCP/IP connection. The server application manages a single repository that consists of one or more depots. A client application communicates with the server to allow the user to view:

- trees of versioned files
- repository metadata (file history, users, groups, labels, permissions)

Clients also manage local workspaces (local directories) 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. Versioned files are stored on the server in depots of various types, such as:

- local
- stream (Helix Core Server User Guide covers Streams in depth)
- graph, which supports Git repos (see the Helix4Git Administrator Guide)

Figure 4-1 **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.

> **Tip**
> To learn about advanced options for servers and services, such as commit, edge, proxy, broker, and replica, see "Deployment architecture" on page 369.

If you want a way to work disconnected from a shared central server, see also "Centralized and distributed architecture" in Using Helix Core for Distributed Versioning (DVCS).

Basic workflow

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

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

   Review installation pre-requisites in "Planning the installation" on page 29.

2. Download and install Helix server.

   See "Installing the server" on page 28.

3. Start the server.

   See the appropriate section on starting the server in "Installing the server" on page 28.

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

See "When should protections be set?" on page 148.

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 "Configuring the server" on page 71.

6. Define additional depots if needed.

   See "Working with depots" on page 109.

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

   See "Creating standard users" on page 231.

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

   See "Securing the server" on page 122.

9. Back up the server.

   See "Securing the server" on page 122.

10. Monitor server performance and resource use.

    See "Monitoring the server" on page 199.

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

    See "Managing the server and its resources" on page 225.

12. Tune the server to improve performance.

    See "Tuning Helix server for performance" on page 245.

13. Customize Helix server by extending job definitions.

    See "Customizing Helix server: job specifications" on page 274.

14. Customize Helix server using trigger scripts.

    See "Triggers and Extensions" on page 284.

Administrative access

Helix server security depends on the security level that is set and on how authentication and access privileges are configured; these are described in "Securing the server" on page 122. Access levels relevant for the administrator are `admin` and `super`:

- `admin` grants permission to run Helix server 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 Helix server 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 *Helix Core P4 Command Reference* indicate the access level needed to execute that command.

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

```
$ p4 protect
```

as soon as possible to define a Helix server superuser.

Naming Helix server objects

As you work with Helix server, 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.
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 Helix server MacBook Pro).
User	The OS user.

Verifying files by signature

Helix server 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 Helix server database for that file revision, Helix server displays the characters **`BAD!`** after the signature, and you should contact Perforce Technical Support.

> **Tip**
> We recommend that administrators perform `p4 verify` weekly, rather than nightly. For large installations, the verification of files:
>
> - takes considerable time to run
> - puts the server under heavy load, which can impact the performance of other Helix server commands

Installing the server

This chapter describes how to install the Perforce service or upgrade an existing installation for connected clients.

- For the list of supported operating systems, see the Release Notes at https://www.perforce.com/perforce/r20.2/user/relnotes.txt, where `r20.2` represents the version number, such as 2020.2
- For the list of supported operating systems for Linux package-based installation, see "Linux package-based installation" on page 35.
- 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.
- For information on how to install a server that supports clients who want to work disconnected, see the Installation chapter of *Using Helix Core Server for Distributed Versioning*.

Planning the installation

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

Network

Helix server can run over any TCP/IP network. For remote users or distributed configurations, Helix server 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 "Tuning Helix server for performance" on page 245.

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 248.

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, Helix server 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 "Tuning Helix server for performance" on page 245.

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 "Filesystems" on page 248.

Filesystem performance

Helix server is judicious with its use of disk I/O. Helix server 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 Helix Core 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.

Helix server does not recommend any specific hardware configuration or file system. Linux servers tend to have the best performance because of Linux’s asynchronous directory updating. However, a Linux server might 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 and the underlying storage hardware. Helix server 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 (particularly on Linux and FreeBSD), we recommend that the Helix server database, depot, and journal files be stored on disks that are local to the machine running the Helix Core server process or that they be stored on a low-latency SAN device.

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

Separate physical drives for server root and journal

We recommend that the **`P4ROOT`** directory (that is, the directory containing your database and versioned files) be 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.

See "Backup and recovery" on page 175 and in "Location of db.* files, journal, and depot files" on page 245.

Protections and passwords

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

```
$ p4 protect
```

as soon as possible to define a Helix server superuser. To learn more about how `p4 protect` works, see "Authorizing access" on page 147.

Without passwords, any user is able to impersonate any other Helix server user, either with the **`-u`** flag or by setting `P4USER` to an existing Helix server user name. Use of Helix server passwords prevents such impersonation. See "Passwords" in the *Helix Core Server User Guide*.

To set (or reset) a user’s password, either

- use `p4 passwd` ***`username`*** (as a Helix server 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 Helix server 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 Helix Core 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 the chapter on "Securing the server" on page 122, including "Recommended settings to configurables for security " on page 122.

Protocol levels of server and client by server release number

As part of the initial communication, the client and server negotiate which protocol levels to use so they can understand each other.

The logs might show a protocol level instead of a release number. The "Table of protocols by release" on the facing page shows the server protocol number and client protocol numbers.

Example of server protocol level

When launching a 2020.1 Helix Server with RPC tracing set to level 5 (`-vrpc=5`), the server protocol level of `50` appears as the value of the `server2` variable.

```
p4d -r . -p 1666 -vrpc=5
RpcSendBuffer xfiles = 5
RpcSendBuffer server = 3
RpcSendBuffer server2 = 50
RpcSendBuffer revver = 7
RpcSendBuffer nocase = Perforce Server starting...
```

> **Note**
> The server protocol level is unrelated to the Upgrades counter of the Helix Server Database Schema.

Setting the client protocol level

A later client protocol might provide information in a format that earlier client applications are unable to handle without additional parsing or translation. When using the Perforce Helix API, you can set this value to your required protocol level:

Using the Perforce Helix API
`ClientAPI::SetProtocol("api","<value>");`
See the example code at ClientApi::SetProtocol(char *, char *) in *Helix Core C/C++ Developer Guide*

Note
By default, the Perforce Helix API uses a protocol level matching the version of the API libraries. Therefore, it is often not necessary to set the protocol level value explicitly.

Table of protocols by release

Note
A client release might use the client protocol associated with an earlier server release. For example, the 2020.2 P4V client uses the client protocol 88, which is associated with the 2020.1 server release.

Release	Server Protocol	Client Protocol
2021.1	52	90
2020.2	51	89
2020.1	50	88
2019.2	49	87
2019.1 patch 2	48	-
2019.1	47	86
2018.2	46	85
2018.1	45	84
2017.2	44	83
2017.1	43	82
2016.2	42	81
2016.1	41	80
2015.2	40	79
2015.1	39	78
2014.2	38	77
2014.1	37	76
2013.3	36	75
2013.2	35	74
2013.1	34	73
2012.2	33	72
2012.1	32	71
2011.1	31	70
2011.1 early	-	69 (action resolves disabled)
2008.1	25	63
2007.3	24	62
2007.2	23	61
2006.2	22	60
2006.1	21	59
2005.2	20	58
2005.1	19	-
2004.2	18	-
2004.1	-	57
2003.2	17	56
2003.1	16	-
2003.1 early	15	-
2002.2	14	55
2002.1	13	54
2001.2	12	52
2001.1	11	51
2000.2	10	-
2000.1	9	-
99.2	8	8
99.1	7	7
99.1 early	6	6
98.2	5	5
98.2 early	4	-
98.1	-	4
97.3	3	3

Release	Server Protocol	Client Protocol
2010.2	30	68
2010.1	29	67
2009.2	28	66
2009.1	27	65
2008.2	26	64
97.2	2	2
97.1	1	1

Linux package-based installation

> **Note**
> Helix server requires two executables:
>
> - the Helix Core server, also referred to as the Perforce service (`p4d`)
> - at least one Helix Core client application, such as the Command-Line Client (`p4`)
>
> The Helix server and applications are available on the Perforce web page for Downloads.

Distribution packages simplify the installation, update, and removal of software because the tools that manage these packages are aware of the dependencies for each package.

The Perforce service is available in two distribution package formats:

- Debian (`.deb`) for Ubuntu systems
- RPM (`.rpm`) for CentOS, RedHat Enterprise Linux (RHEL), and SUSE

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

- Ubuntu 12.04 LTS (Precise), 14.04 LTS (Trusty), 16.04 LTS (Xenial), 18.04 LTS (Bionic), 20.04 (Focal)
- CentOS or Red Hat 6.x, 7.x, 8.x
- SUSE Linux Enterprise Server 11, 12, 15

Prerequisites

- root level access to the server that will host your Perforce service
- knowing whether you will need to stop and restart your server or not - see "Release and license information: adding or updating" on page 72.

- reading:
  - "Case sensitivity and multi-platform development" on page 82
  - "Setting up and managing Unicode installations" on page 84

  to prepare for the choices you must make during "Installation" below

Installation

This topic assumes you have met the "Prerequisites" on the previous page.

The Helix server is divided into multiple packages, so you can install 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 (`p4`, which is distributed as `helix-cli`), the service controller (`p4dctl`), 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.

Package installation requires sudo or root level privileges.

Verify the Public Key

To ensure you have the correct public key for installing Perforce packages, verify the fingerprint of the Perforce public key against the fingerprint shown below.

1. Download the public key at https://package.perforce.com/perforce.pubkey
2. To obtain the fingerprint of the public key, run:
   ```
   gpg --with-fingerprint perforce.pubkey
   ```
3. Verify that it matches this fingerprint:
   ```
   E581 31C0 AEA7 B082 C6DC 4C93 7123 CB76 0FF1 8869
   ```

Follow the instructions that apply to you:

- "For APT (Ubuntu) " on the facing page
- "For YUM (Red Hat Enterprise Linux or CentOS)" on the facing page
- "For SUSE Linux Enterprise Server" on the facing page

For APT (Ubuntu)

1. Add the Perforce packaging key to your APT keyring. For example,
   ```
   wget -qO - https://package.perforce.com/perforce.pubkey | sudo
   apt-key add -
   ```
2. Add the Perforce repository to your APT configuration.
   Create a file called `/etc/apt/sources.list.d/perforce.list` with the following line:
   ```
   deb http://package.perforce.com/apt/ubuntu {distro} release
   ```
   Where `{distro}` is replaced by one of the following: `precise`, `trusty`, `xenial`, `bionic`, or `focal`.
3. Run `apt-get update`
4. Install the package by running `sudo apt-get install helix-p4d`

You can also browse the repository and download a Deb file directly from https://package.perforce.com/apt/

See "Post-installation configuration" on the next page.

For YUM (Red Hat Enterprise Linux or CentOS)

1. Add Perforce's packaging key to your RPM keyring:
   ```
   sudo rpm --import https://package.perforce.com/perforce.pubkey
   ```
2. Add Perforce's repository to your YUM configuration.
   Create a file called `/etc/yum.repos.d/perforce.repo` with the following content:
   ```
   [perforce]
   name=Perforce
   baseurl=http://package.perforce.com/yum/rhel/{version}/x86_64
   enabled=1
   gpgcheck=1
   ```
   where `{version}` is either 6 for RHEL 6 or 7 for RHEL 7
3. Install the package by running `sudo yum install helix-p4d`

- You can also browse the repository and download an RPM file directly: https://package.perforce.com/yum/

See "Post-installation configuration" on the next page.

For SUSE Linux Enterprise Server

1. Add Perforce's packaging key to your RPM keyring:
   ```
   sudo rpm --import http://package.perforce.com/perforce.pubkey
   ```

2. Add the Perforce repository.

   `sudo zypper addrepo http://package.perforce.com/yum/rhel/7/x86_64/ helix`

3. Install the package by running `sudo zypper install helix-p4d`

- You can also browse the repository and download an RPM file directly: https://package.perforce.com/yum/

See "Post-installation configuration" below.

Post-installation configuration

After the `helix-p4d` package has been installed, additional configuration is required to create a Helix server.
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 Helix server.

   > **Note**
   > If you already have a Helix server 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 when starting and stopping the service with `"Helix Core Server Control (p4dctl)" on page 511`.

   This name is also used to set the Perforce **`serverid`** attribute on this **`p4d`** server, to distinguish it from other **`p4d`** servers that might 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 84 for information.

d. The Case Sensitivity for the server:

   This is on by default.

   See "Case sensitivity and multi-platform development" on page 82 for information.

e. The Server Address (`P4PORT`):

   This specifies the host and port where the Helix server should listen, and whether to communicate in plaintext or over SSL. For more information, see "Communicating port information" on page 45.

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 151.

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 "Helix Core Server Control (p4dctl)" on page 511.

Updating

> **Important**
> The package update commands with `apt-get`, `yum,` or `zypper` 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 Helix server in a precarious state. Make sure to read and understand the entire process before updating any packages.

1. Review the update (upgrading) process.
   a. See "Upgrading the server" on page 56.
   b. Packages for Linux help you accomplish only **specific steps** from the **general process**. If you are attempting to update your Helix server 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 Helix server 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 Helix server 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 Helix server 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:

   - **For Ubuntu:**

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

   - **For CentOS/RHEL:**

```
$ yum --showduplicates list helix-p4d
```

- **For SUSE Linux Enterprise Server:**

```
$ sudo zypper search -s 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:

- **For Ubuntu:**

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

- **For CentOS/RHEL:**

```
$ sudo yum install helix-p4d
```

- **For SUSE Linux Enterprise Server:**

```
$ sudo zypper install helix-p4d
```

> **Important**
> Failure to complete all update steps could result in continued downtime for your Helix server.

Linux non-package installation

Linux non-package installation: quick example

The quickest way to get started:

1. Make a directory for your installation:
   ```
   mkdir newinstall
   ```
2. Navigate to the newly-created directory:
   ```
   cd newinstall
   ```
3. Go to https://www.perforce.com/downloads/helix-core-p4d
   a. Under **Family**, click **Linux**
   b. Under **Platform**, click **Linux**
   c. Click **Download**
4. Copy the **p4d** file in your newly-created directory.
5. Give the OS user execution permission for the **p4d** file:
   ```
   chmod 755 p4d
   ```
6. Configure the OS environment to have a Perforce user:
   ```
   export P4USER=perforce
   ```
7. Configure the OS environment to have a Helix server port:
   ```
   export P4PORT=localhost:1666
   ```
8. Invoke the server executable, `p4d`, while specifying the current directory with `` `pwd` `` enclosed in backticks and the port as `-p 1666`:
   ```
   ./p4d -r `pwd` -p 1666 -J journal -L log -d
   ```
   The output is:
   ```
   Perforce db files in '/home/bruno/newinstall' will be created
   if missing...
   Perforce Server starting...
   ```
9. Go to https://www.perforce.com/downloads/helix-command-line-client-p4
   a. Under **Family**, click **Linux**
   b. Under **Platform**, click **Linux (x64)**
   c. Click **Download**
10. Copy the **p4** file to the same directory where you copied the **p4d** file.
11. Give the OS user execution permission for the **p4** file:
    ```
    chmod 755 p4
    ```
12. Issue the p4 info command as follows, with `./` before `p4 info`:
    ```
    ./p4 info
    ```

13. From the output, write down for future use the `Server address` value:

```
User name: perforce
Client name: linux-bruno
Client host: linux-bruno
Client unknown.
Current directory: /home/bruno/newinstall
Peer address: 127.0.0.1:12345
Client address: 127.0.0.1
Server address: localhost:1666
Server root: /home/bruno/newinstall
Server date: 2018/11/14 15:18:55 -0800 PST
Server uptime: 00:00:09
Server version: P4D/LINUX26X86_64/2018.2/1234567 (2018/11/02)
Server license: none
Case Handling: sensitive
```

14. Download the Helix Visual Client application that has a graphical user interface, P4V, from P4V Download. Depending on your flavor of Linux, the steps might be similar to:
    a. Extract `p4v.tgz`
    b. Install the application.
    c. Invoke the executable: `./p4v`
15. As shown in the "Connecting with P4V" video, connect the client to the "remote" server by using the `Server address` from Step 13:
16. Watch the videos on "Setting up Workspaces in P4V" and "Basic Operations in P4V".

General considerations for Linux non-package installation

> **Note**
> Helix server requires two executables:
>
> - the Helix Core server, also referred to as the Perforce service (`p4d`)
> - at least one Helix Core client application, such as the Command-Line Client (`p4`)
>
> The Helix server and applications are available on the Perforce web page for Downloads.

Although you can install `p4` and `p4d` in any directory, on Linux the Helix server 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 Helix server 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 Helix server user account that has been created for the purpose of running the Perforce service.

> **Note**
> To maximize performance, configure the server root (`P4ROOT`) to reside on a local disk and not an NFS-mounted volume. It is best to place metadata and journal data on separate drives.
>
> Helix server applications (such as `p4`, the Helix server Command-Line Client) work with client workspaces on NFS-mounted drives, such as client workspaces located in users' home directories.

Creating a Helix server root directory

This topic assumes that you have download the `p4` and `p4d` binaries and have made `p4` and `p4d` executable.

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`.

> **Note**
> `p4d` is the only process that uses the `P4ROOT` variable. Helix Core client applications never use the `P4ROOT` directory or environment variable.

Because all Helix server 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 30 for information about diskspace requirements.

Do not run p4d as root

The Perforce service requires no privileged access. Do NOT run `p4d` as `root` or any other privileged user. See "Running the Helix server (p4d) as an unprivileged user" on page 46.

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 Helix server applications which port to connect to

The `p4d` service and Helix server applications communicate with each other using TCP/IP. When `p4d` starts, it listens (by default) for plaintext connections on port `1666`. Helix server 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 Helix server" on page 123 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 the next page for more information about IPv6 and IPv4 transports.

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

Communicating port information

Helix server 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 Helix server 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:server1:3435`	Helix server applications connect in plaintext to the Helix server on host `server1` listening on port `3435`.
`tcp64:server1:3435`	Helix server applications connect in plaintext to the Helix server on host `server1` 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`	Helix server applications connect via SSL to the Helix server on host `example.org` listening on port `1818`.
`<not set>`	Helix server applications connect to the Helix server 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 and trusted Helix 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, Helix server supports connectivity over IPv6 networks as well as over IPv4 networks. For details, see P4PORT in Helix Core Server Administrator Guide.

> **Note**
> In multi-server environments, the net.rfc3484 configurable, when set server-side, also controls the behavior of host resolution when initiating communications for server-to-server, proxy, or broker.

Running the Helix server (p4d) as an unprivileged user

Helix server 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

Under a normal installation, the Perforce service runs on Linux 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 Helix server
- `/usr/local/bin` is the directory holding your `p4d` binary
- `p4droot` is the root directory (`P4DROOT`) to use for this Helix server (for example, `/usr/local/p4d`)
- `username` is the UNIX user name to use for running this Helix server
- `nnnn` is the port number for this Helix 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, an organization might have a several test servers running on UNIX, each defined as an `inetd` service with its own port number.

There are caveats with this method:

- `inetd` can disallow excessive connections, so a script that invokes several thousand `p4` commands, each of which spawns a `p4d` server via `inetd`, might 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 because the `p4d` executable is run each time. Disabling the server requires modifying `/etc/inetd.conf` and restarting `inetd`.
- To use Helix server with this license, you need to request a server license that does not specify a port. Contact https://www.perforce.com/support/request-support.

> **Note**
> For information about using `systemd` to launch services and daemons at boot time, see the Support Knowledgebase article, Example systemd Perforce Service File.

Starting the Helix server

After you set `p4d`'s `P4PORT` and `P4ROOT` environment variables, start the server 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 **Starting the Helix server**

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 "Backup and recovery" on page 175. To enable SSL support, see "Using SSL to encrypt connections to a Helix server" on page 123. A complete list of flags is provided in the "Helix Core server (p4d) Reference" on page 499.

For information about the files that have been installed, see "Installed files" on page 54.

Stopping the Helix server

To shut down the Helix server, use the command:

```
$ p4 admin stop
```

Only a Helix server superuser can use `p4 admin stop`.

If you are having problems stopping your Helix Server, email support@perforce.com.

Restarting a running Helix server

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

```
$ p4 admin restart
```

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

Windows installation

Windows installation: quick example

The quickest way to get started:

1. Log on to Windows with Administrator privileges to install the Helix server.
2. Download the Helix core server software from https://www.perforce.com/downloads/helix-versioning-engine-p4d
3. Install the Perforce server using the downloaded installer binary.
   a. Choose the features to install - `Server (P4D)` and `Command-Line Client (P4)`.
   b. Choose the default Port Number 1666 or specify another port number.
   c. Choose the default server location or specify a new location.
   d. When prompted for Client Configuration, type in `Server field <machine name>:1666` where `<machine name>` is the name of your machine, and type in the `User Name` field the Helix username you want to use.
   e. The Windows services applet can be used to stop and start the `Perforce` service (the Helix Versioning Engine).
4. Verify that the Helix server is running by issuing in a command window `p4 -p <machine name>:1666 info`
5. To connect to your new Helix server, download Helix P4V from https://www.perforce.com/downloads/helix-visual-client-p4v
6. Install the Helix Visual Client.
7. As shown in the "Connecting with P4V" video, connect the client to the "remote" server.
8. Watch the videos on "Setting up Workspaces in P4V" and "Basic Operations in P4V".

> **Note**
> If you see the following error message:

> "Helix Versioning Engine cannot be installed because setup has detected that this machine is already configured for distributed version control."
>
> Perform these steps:
>
> 1. Locate the `p4d.exe` file in the DVCS folder.
> 2. Remove it or rename it.
> 3. Run the installer.
>
> See also the Support Knowledgebase article, Error Installing Helix Server on Windows.

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 except for the 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:

- `p4s.exe` invokes a service
- `p4d.exe` invokes a server

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`.
>
> See the Knowledge Base article "Changing the user account the Windows service runs under".

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 Helix server installer (`helix-versioning-engine-x64.exe` or `helix-versioning-engine-x86.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 Helix server 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 Helix server as a service under Windows, the service starts whenever the machine boots.

Stopping the service with C:\> svcinst stop -n "Perforce"

Important
To stop the service, the Helix server superuser issues the command:

```
C:\> svcinst stop -n "Perforce"
```

Warning
To avoid the risk of problems, do not attempt to stop the service in any other way.

Note
If you are using a 32-bit version of the Helix server, you might need to download the `svcinst.exe` tool from ftp://ftp.perforce.com/perforce/tools/svcinst/bin.ntx86 to your Helix server root directory before you issue the command:

```
C:\> svcinst stop -n "Perforce"
```

If you are having problems stopping your Helix Server, email support@perforce.com.

Multiple Perforce services under Windows

By default, the Helix server installer for Windows installs a single Helix Core server as a single service. If you want to host more than one Helix server installation on the same machine (for instance, one for production and one for testing), you can either manually start Helix 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 "Configuration parameter precedence" on the facing page.

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 Helix server installer, and can be found in your Helix server root.)
4. Set up the environment variables and start the new service.

We recommend that you install your first Perforce service using the Helix server 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 **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 Helix server installation you installed in `C:\perforce`.

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 `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.

Configuration parameter precedence

Tip
You can specify client settings such as port, user, and workspace names by using any of the following:

1. On the command line, using options.
2. In the configuration file(s) specified by a P4CONFIG environment variable, where each config file can be specific to a workspace.
3. In the P4ENVIRO configuration file, which is for variables that remain constant for all the workspaces on a given computer.
4. User environment variables.
5. System environment variables (on Windows, system-wide environment variables are not necessarily the same thing as user environment variables)
6. In the user registry or settings set by issuing the `p4 set` command.
7. In the system registry or system settings set by issuing the `p4 set -s` command.

where

- Command line overrides P4CONFIG
- P4CONFIG overrides P4ENVIRO, and so on.

The output of `p4 set` lists the values of the variables (and if a given variable was set by `config`, `enviro`, `set`, or `set -s`).

Support for long file names

Support for long file names is enabled by default in Helix server versions 2015.2 or later. For older versions of Helix 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 Helix server host:

- Database files
- The Journal file
- The Helix server binary

By default, the database files and the Journal file are placed in the server root directory (P4ROOT) of the Helix Core server.

Eventually, as users and administrators work with Helix Server, other files are added to the Helix Server root directory: archived files (also called depot files), checkpoint and rotated journal files, and log files (P4LOG).

Operating system	Location of Helix server binary
Linux	Where the administrator puts it. Usually **/usr/local/bin/p4d** or, if installed via packages, **/opt/perforce/bin/p4d**.
Mac OS X	Where the administrator puts it. Usually **/usr/bin/p4d** or **/user/local/bin/p4d**.

Operating system	Location of Helix server binary
Windows	Where the administrator puts it. By default **`C:\Program Files\Perforce\Server\p4d`**.

Upgrading the server

Note

- This chapter describes how to upgrade an existing installation for connected clients. For information on how to install a server that supports clients who want to work disconnected, see the "Installation" chapter of *Using Helix Core Server for Distributed Versioning*.
- The examples in this chapter apply to both Windows and Linux/MacOS installations.

Older Helix Core applications continue to work with newer versions of Helix Core server. However, to enable your users to benefit from the features introduced in subsequent versions of the server, you must upgrade the server.

Windows

Use the installer to install the new version.

Important
When planning the upgrade:

- To upgrade Helix Server to a newer version, your Helix Server license file must be current.
- You *must* back up your Helix server installation (see "Backup procedure" on page 182) as part of any upgrade process.
- If you have an installation of Helix4Git, upgrade your Helix4Git installation before upgrading your Helix Core servers. See Upgrading Git Connector in *Helix4Git Administrator Guide*.
- Before you upgrade the Helix server, read the release notes associated with your upgraded installation.
- If running the Helix Versioning Engine as a service (`p4s.exe`) and upgrading manually rather than using the Windows installer, make sure that binary is also updated by copying `p4d.exe` to `p4s.exe`.
- If you have a large number of ktext files (`+k` and `+ko` text files with RCS keyword expansion), contact Perforce Support for guidance on making the upgrade process faster.

Linux

Note
It is possible for more than one server to run on the same host machine. On a Linux host, if you want to make sure all such servers use a specific version and do not upgrade to higher versions, consider this approach.

APT policy with `Pin`	`more /etc/apt/preferences.d/p4d`
	`Package: helix-p4d`
	`Pin: version 2019.2.*`
	`Pin-Priority: 1000`

Yum	See https://www.cyberciti.biz/faq/centos-redhat-fedora-yum-lock-package-version-command/ The `--exclude` directive allows for wildcards, whereas the `yum versionlock` command will not allow patch upgrades.

Alternatively, you can install a specific base package in a way that allows the package manager to automatically install the latest patch. For example,

```
apt-get install helix-p4d-base-r19.2
sudo update-alternatives --config helix-p4d
```

If you want to allow for version differences between the p4d services running on the same Linux host, you can use "Helix Core Server Control (p4dctl)" on page 511 and set the version for each specific server. For example,

- on server1, you might update the execute parameter in the p4dctl config to point to `/opt/perforce/sbin/p4d.20.2`
- on server2, you might update the execute parameter in the p4dctl config to point to `/opt/perforce/sbin/p4d.19.2`

Tip

If the server was created by `configure-helix-p4d.sh`, and hence is managed by "Helix Core Server Control (p4dctl)" on page 511, upgrade the database with this command: `p4dctl exec -t p4d <service name> -- -xu`

This ensures the `-xu` is run with the correct user and environment.

Important

When planning the upgrade:

- To upgrade Helix Server to a newer version, your Helix Server license file must be current.
- You *must* back up your Helix server installation (see "Backup procedure" on page 182) as part of any upgrade process.
- If you have an installation of Helix4Git, upgrade your Helix4Git installation before upgrading your Helix Core servers. See Upgrading Git Connector in *Helix4Git Administrator Guide*.
- Before you upgrade the Helix server, read the release notes associated with your upgraded installation.

- If running the Helix Versioning Engine as a service (`p4s.exe`) and upgrading manually rather than using the Windows installer, make sure that binary is also updated by copying `p4d.exe` to `p4s.exe`.
- If you have a large number of ktext files (`+k` and `+ko` text files with RCS keyword expansion), contact Perforce Support for guidance on making the upgrade process faster.

Tip

- To check on the status of upgrade steps on a single server, use p4 upgrades
- To check on the status of the specified upgrade step on a server and its upstream servers, use `p4 upgrades -g`

Verifying files before and after a server upgrade

Both before and after you upgrade the server, we recommend that you use the p4 verify command:

1. Before the upgrade, verify the integrity of:
   1. submitted file revisions by running:

      $ **p4 verify -q //...**
   2. shelved files (if any) by running:

      $ **p4 verify -qS //...**
2. Take a checkpoint.
3. Copy the checkpoint and your versioned files to a safe place.
4. Perform the server upgrade.
5. After the upgrade, verify the integrity of
   1. submitted file revisions by running:

      $ **p4 verify -q //...**
   2. shelved files (if any) by running:

      $ **p4 verify -qS //...**

See also p4 verify and p4 shelve in *Helix Core P4 Command Reference*.

From 2019.1 or later to latest (single server)

These instructions assume you have a single server. If not, see "From 2019.1 or later to latest (multi-server environment)" on the facing page.

Note

- A proxy or broker only needs to get the latest binary.
- You do NOT need to run `p4d -xu` against offline db.* files (a set of db.* files periodically brought current by replaying the journal files from the production server).

Prerequisites

- To upgrade Helix Server to a newer version, your Helix Server license file must be current.
- We strongly recommend that you take a checkpoint so that you know you can recover if something goes wrong during the upgrade. (See "Checkpoint files" on page 176.) It is best to take the checkpoint at a time when the end-users are not active.
- Make sure that **journaling is enabled**.

Upgrade Steps

In the following steps, replace

- `[P4ROOT]` with the path to the P4ROOT directory
- `[P4JOURNAL]` with the directory path and name of the journal file

1. Shut down the server by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS account owner of the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`
4. Wait until you see this message:
   `Upgrades will be applied at server startup.`
5. Start the server.
6. Inform your users that they can resume work.

From 2019.1 or later to latest (multi-server environment)

These instructions assume you have a multi-server environment. If not, see "From 2019.1 or later to latest (single server) " on the previous page.

Note

- A proxy or broker only needs to get the latest binary.
- You DO need to run `p4d -xu` against all servers and replicas.

> - You do NOT need to run `p4d -xu` against offline db.* files (a set of db.* files periodically brought current by replaying the journal files from the production server).

Prerequisites

- To upgrade Helix Server to a newer version, your Helix Server license file must be current.
- Make sure that **journaling is enabled** so it can apply the upgrades to other servers.

Upgrade Steps

In the following steps, replace

- `[P4ROOT]` with the path to the P4ROOT directory
- `[P4JOURNAL]` with the directory path and name of the journal file

In the case of commit edge, the "innermost server" is the commit server and the outer servers are edge servers. In the case of a standard server and replicas of that standard server, the "innermost server" is a standard server.

On the outermost server

1. Shut down the server by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS user that runs the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`
4. Wait until you see this message:
   `Upgrades will be applied from '-xu' at upstream server`
5. Restart the server.

On the servers between the outermost server and the innermost server

1. Shut down by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS account owner of the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`
4. Wait until you see this message:
   `Upgrades will be applied from '-xu' at upstream server`
5. Restart the server.

On the innermost server

1. Shut down by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS account owner of the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`
4. Wait until you see this message:
   `Upgrades will be applied at server startup.`
5. As the OS account owner of the Helix server, start the **innermost** server.
6. Inform your users that they can resume work.

From 2013.3 - 2018.2 to latest (single server)

> **Note**
> The new `db.storage` table replaces `db.archmap` to provide a link count for archive files on the server. This allows `+Sn` type files to be lazy copied, which saves disk space. The upgrade also improves the performance of p4 obliterate, especially on servers with significant integration history.

These instructions assume you have a single server. If not, see "From 2013.3 - 2018.2 to latest (multi-server environment)" on page 64.

> **Note**
> - A proxy or broker only needs to get the latest binary.
> - You do NOT need to run `p4d -xu` against offline db.* files (a set of db.* files periodically brought current by replaying the journal files from the production server).

Prerequisites

> **Important**
> - To upgrade Helix Server to a newer version, your Helix Server license file must be current.

1. Time:
   - "Phase 1" on the next page of the upgrade requires that the servers be down for several minutes.
   - "Phase 2" on page 63 of the upgrade runs in the background so users can work. This phase might take several hours. Performance might be slower while the `db.storage` table is being built.

   - Environments with a `db.rev` table larger than 200 GB might take a couple hours to complete.
   - Smaller environments might only take several minutes.

2. Available disk space should be at least equal to the size of the db.rev table. (The journal file can grow to the size of the new `db.storage` table. Typically, the journal is on a separate partition.)
3. Make sure that **journaling is enabled**.

> **Important**
>
> - We strongly recommend that you take a checkpoint so that you know you can recover if something goes wrong during the upgrade. (See "Checkpoint files" on page 176.) It is best to take the checkpoint at a time when the end-users are not active.
> - Let the users know that the server will be down several minutes for maintenance, followed by several hours in which performance might be slower and the following commands are blocked:
>   - fetch
>   - obliterate
>   - push (for DVCS)
>   - reload
>   - storage
>   - unload (for administration)
>   - unsubmit

Upgrade Steps

In the following steps, replace

- `[P4ROOT]` with the path to the P4ROOT directory
- `[P4JOURNAL]` with the directory path and name of the journal file

Phase 1

1. Shut down the server by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS account owner of the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`

4. Wait until you see this sequence of messages complete and expect a pause after the first message:

   ```
   2019.1: building db.storage from db.rev, db.revsh and db.revtx
   [pause]
   2019.1: Adding default namespace to Extension configurations
   Additional upgrades will be applied at server startup.
   ```

   (The upgrades steps for the new version will be executed by the server as it is started)

5. As the OS account owner of the Helix server, start the server. This starts a background process to build the `db.storage` table, a new table in the Helix server schema.

6. Inform your users that they can resume work.

Phase 2

1. To know when the background upgrade process is complete, run p4 storage `-w` on the the innermost server.

2. Wait for p4 storage `-w` to return the following message: "`The storage upgrade process is complete`".

3. To monitor progress, on a separate command-line terminal, run the following command from time to time to verify that the timestamp or filesize is increasing:

   - `ls -l db.storage` for Linux
   - `dir /ta /od` for Windows

If you have a huge number of keyword revisions

If you have a huge number of keyword revisions, creating a digest for each one might lengthen the upgrade time. To make the upgrade "skip" the creation of such digests, follow these steps:

1. Shut down the server by running `p4 admin stop`

2. Install the new version of the `p4d` binary on the system.

3. Set the lbr.storage.skipkeyed configurable, using the new p4d binary:
   `p4d -r` *`[P4ROOT]`* `"-cset lbr.storage.skipkeyed=1"`

4. As the OS user that runs the Helix server, run `p4d -r` *`[P4ROOT]`* `-J` *`[P4JOURNAL]`* `-xu`

5. Start the new server.

6. After "Phase 2" above is completed, run the `p4 storage -U -q //...` command to update the digests of your ktext file revisions. We recommend you do this overnight or on the weekend because it might reduce performance for a considerable time. (See File types in *Helix Core P4 Command Reference*.)

From 2013.3 - 2018.2 to latest (multi-server environment)

> **Note**
> The new `db.storage` table replaces `db.archmap` to provide a link count for archive files on the server. This allows `+Sn` type files to be lazy copied, which saves disk space. The upgrade also improves the performance of p4 obliterate, especially on servers with significant integration history.

These instructions assume you have a multi-server environment. If not, see "From 2013.3 - 2018.2 to latest (single server) " on page 61.

> **Note**
> - A proxy or broker only needs to get the latest binary.
> - You DO need to run `p4d -xu` against all servers and replicas.
> - You do NOT need to run `p4d -xu` against offline db.* files (a set of db.* files periodically brought current by replaying the journal files from the production server).

Prerequisites

1. To upgrade Helix Server to a newer version, your Helix Server license file must be current.
2. Time:
   - "Phase 1" on page 66 of the upgrade requires that the servers be down for several minutes.
   - "Phase 2" on page 67 of the upgrade runs in the background so users can work. This phase might take several hours. Performance might be slower while the `db.storage` table is being built and its contents are being replicated to the outer servers.
     - Environments with a `db.rev` table larger than 200 GB might take a couple hours to complete.
     - Smaller environments might only take several minutes.
3. Available disk space should be at least equal to the size of the db.rev table. (The journal file can grow to the size of the new `db.storage` table. Typically, the journal is on a separate partition.)
4. Make sure that **journaling is enabled** because journal replication is how the new upgrades will be copied to other servers.
5. Be prepared to upgrade all the servers in a single upgrade session.

Important

- When upgrading **from prior to 2019.1 to 2019.1 or later**, we strongly recommend that all servers be upgraded in a single upgrade session.
- We strongly recommend that you take a checkpoint so that you know you can recover if something goes wrong during the upgrade. (See "Checkpoint files" on page 176.) It is best to take the checkpoint at a time when the end-users are not active.
- Let the users know that all the servers will be down several minutes for maintenance, followed by several hours in which performance might be slower and the following commands are blocked:
  - fetch
  - obliterate
  - push (for DVCS)
  - reload
  - storage
  - unload (for administration)
  - unsubmit

Upgrade Steps

In the following steps, replace

- `[P4ROOT]` with the path to the P4ROOT directory
- `[P4JOURNAL]` with the directory path and name of the journal file

In the case of commit edge, the "innermost server" is the commit server, and the outer servers are edge servers. In the case of a standard server and replicas of that standard server, the "innermost server" is a standard server.

- The upgrade proceeds from the outermost server toward the innermost server
- Restarting will proceed from innermost server towards the outermost server.

On the outermost server

1. Shut down the server by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS account owner of the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`

4. Wait until you see this sequence of messages complete and expect a pause after the first message:
   ```
   2019.1: building db.storage from db.rev, db.revsh and db.revtx
   [pause]
   2019.1: Adding default namespace to Extension configurations
   Additional upgrades will be applied from '-xu' at upstream
   server.
   ```
   (The upgrades steps for the new version will be replicated outward from the innermost server)
5. **Leave this server down.**

On the servers between the outermost server and the innermost server

1. Shut down by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS account owner of the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`
4. Wait until you see this sequence of messages complete and expect a pause after the first message:
   ```
   2019.1: building db.storage from db.rev, db.revsh and db.revtx
   [pause]
   2019.1: Adding default namespace to Extension configurations
   Additional upgrades will be applied from '-xu' at upstream
   server.
   ```
   (The upgrades steps for the new version will be replicated outward from the innermost server)
5. **Leave this server down.**

On the innermost server

Phase 1

1. Shut down by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. As the OS account owner of the Helix server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`
4. Wait until you see this sequence of messages complete and expect a pause after the first message:
   ```
   2019.1: building db.storage from db.rev, db.revsh and db.revtx
   [pause]
   2019.1: Adding default namespace to Extension configurations
   Additional upgrades will be applied at server startup.
   ```

5. As the OS account owner of the Helix server, start the **innermost** server. This starts a background process to build the `db.storage` table, a new table in the Helix server schema.
6. As the OS account owner of the Helix server, start the servers **between the innermost server and the outermost server**.
7. As the OS account owner of the Helix server, start the **outermost** server.
8. Inform your users that they can resume work.

Phase 2

1. To know when the background upgrade process is complete, run p4 storage `-w` on the the innermost server.
2. Wait for p4 storage `-w` to return the following message: "`The storage upgrade process is complete`".

To monitor progress, on a separate command-line terminal, run the following command from time to time to verify that the timestamp or filesize is increasing:

- `ls -l db.storage` for Linux
- `dir /ta /od` for Windows

If you have a huge number of keyword revisions

If you have a huge number of keyword revisions, creating a digest for each one might lengthen the upgrade time. To make the upgrade "skip" the creation of such digests, **on the commit server**, follow these steps:

1. Shut down the server by running `p4 admin stop`
2. Install the new version of the `p4d` binary on the system.
3. Set the lbr.storage.skipkeyed configurable, using the new p4d binary: `p4d -r [P4ROOT] "-cset lbr.storage.skipkeyed=1"`
4. As the OS user that runs the Helix Server, run `p4d -r [P4ROOT] -J [P4JOURNAL] -xu`
5. Start the new server.
6. After "Phase 2" above is completed, run the `p4 storage -U -q //...` command to update the digests of your ktext file revisions. We recommend you do this overnight or on the weekend because it might reduce performance for a considerable time. (See File types in *Helix Core P4 Command Reference*.)

From prior to 2013.3

Follow these instructions if your old version is 2013.2 or earlier.

Important

- Compared to earlier versions, the 2013.3 version (and later) contains major changes to the database implementation. These changes allow for increased concurrency and scalability, and removes the 16TB size limit for `db.*` database files.
- Although the `db.*` database file format has changed, the checkpoint and journal file formats are identical. **To upgrade from 2013.2 or earlier to version 2013.3 or later**, you *must* restore the database from a checkpoint.
- To upgrade Helix server to a newer version, your license file must be current.

To restore from a checkpoint

1. Stop the Perforce service (`p4 admin stop`).
2. Make a checkpoint and back up your old installation. (see "Backup procedure" on page 182)
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 by using `mv` (Linux) or `move` (Windows). For example:

   ```
   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 your/P4ROOT/directory -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 absolute/path/to/message.txt
   ```

See "Localizing server error messages" on page 86 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. Follow the steps at

    - "From 2013.3 - 2018.2 to latest (single server) " on page 61 or
    - "From 2019.1 or later to latest (multi-server environment)" on page 59

Patching the server

To ensure you have the latest fixes, we recommend that you install the latest patch available for your release at the Software Release Index. Patch installation is quick, but does involve the Helix Server being stopped briefly.

See the instructions below for your type of installation.

Linux package

No need to manually stop and restart the Helix Core server.

Use `apt update`, then `apt upgrade`, and the server will restart with the latest patch.

Linux (or MacOS) manual installation

Download the patched version of the P4D executable from ftp.perforce.com/perforce

Stop the Helix Server.

Overwrite the existing P4D executable with the patch version.

Start the Helix server.

Windows

Stop the Helix Server.

Go to ftp.perforce.com/perforce and do one of the following:

- download the installer, then run it
- download the patched version of the P4D executable, overwrite the existing P4D executable and copy that to `p4s.exe`

Start the Helix server.

Perforce Workshop Server Deployment Package (SDP)

See the SDP project at https://swarm.workshop.perforce.com/projects/perforce-software-sdp/

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.

See also the `p4 configure` command and the server, client, and proxy Configurables in the *Helix Core P4 Command Reference* as well as the command-line output of **`p4 help configurables`**.

> **Tip**
> **`p4 configure show`** displays the current configuration of this server.

Release and license information: adding or updating

License by standard users

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 multi-server development" on page 115) 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.

Limits for unlicensed use depend on the release

2019.2 patch 14, 2020.1 (patch 8), 2020.2 (patch 6), 2021.1 (patch 1), and later	2016.1 and later	prior to 2016.1
Unlimited number of files for 5 users and 20 client workspaces, or Unlimited number of users and workspaces for up to 1,000 files	Unlimited number of files for 5 users and 20 client workspaces, or Unlimited number of users and workspaces for up to 1,000 files	Unlimited number of files for 20 users and 20 client workspaces, or Unlimited number of users and workspaces for up to 1,000 files
Three repos for Helix4Git, which leverages the power and scale of Helix server for large binary assets while allowing you to manage Git repos natively.		

Important
Licensing for replica, edge and standby servers:

Replica servers that are not going to be used for failover and edge servers do not require their own license if they have Helix Core server (P4D) version 2013.2 or later.

Standby servers and replicas that might be required to take over from a master server do require their own license file. This can be obtained by filling out the form at https://www.perforce.com/support/duplicate-server-request.

License information

You can update an existing license file without stopping Perforce by using the `p4 license` command. See "Adding or updating the license file" below.

- If the service is running, any user can use `p4 info` to view basic licensing information.

Tip
We recommend that you hide sensitive information from unauthorized users of `p4 info` by setting the dm.info.hide configurable.

- 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`.

Adding or updating the license file

Note
When you receive your license file, its name might include the license host, an IP address or MAC address, or other identifying information.

Rename the new license file to `license.txt` before you copy it into the P4ROOT directory.

Important
If you are licensed for Helix Core version 2021.1 patch 1 or greater, Helix4Git licensing is included at no extra charge.

Otherwise, if you have purchased Helix4Git in addition to Helix Core, you will receive:

- The email for Helix4Git, which contains the license file that enables both Helix Core and Helix4Git. You will install this license file.
- The email for Helix Core, contains a license file that does NOT enable Helix4Git. Do NOT install this license file, but do keep it for your records.

To add or update the license file, use one of the following:	Valid for ...
Copy the license file to the P4ROOT directory. See "License file in the P4ROOT directory" below	■ Server ■ Any replica that you want to enable to become a master through failover
Issue the p4 license command. See "p4 license command" on the facing page	Server
Use P4Admin. See "Helix Visual Client (P4V) Administration tool" on the facing page	Server

License file in the P4ROOT directory

1. Copy the new license file over any existing license in the `P4ROOT` directory.
2. Determine whether you need to restart the server or not:

Stop and restart the server if ...	No server restart required if ...
Any of the following are true: ■ This is the first time that the license file is being added to the master server `P4ROOT` directory ■ Server's IP address or MAC address changes ■ Port number in the license file changes ■ You are adding a server license to a previously unlicensed replica to prepare for a possible failover	All of the following are true: ■ Server's IP address or MAC address remains as-is ■ Port number in the license file remains as-is ■ Any unlicensed replica remains unlicensed, and thus is not prepared to become the new master in case of a failover **Note** An edge server is a filtered replica and therefore not eligible to be a standby server.

If you need to stop and restart the server ...

	Windows	Linux/UNIX/Mac
To stop the server	`p4 -u User -p Server:Port admin stop`	
To restart the server	Open an administrator command prompt and enter **`net start perforce`** or enter **`services.msc`**, go to the Services Management Console, find **`Perforce`**, and select **`Start`**.	Run the server startup script you usually use. If you do not have a Perforce startup script, the command to start the server in Daemon Mode in the specified P4ROOT location might resemble the following: **`p4d -r /specify/path/to/P4ROOT -d`**

p4 license command

If a valid license file is already in the server root directory, as a **`super`** user, you can update it.

1. Display your current license with **`p4 license -o`**.
2. Install your new license with **`cat license.txt | p4 license -i`** or **`p4 license -i < license.txt`**.

> **Tip**
> If the server IP address, MAC address, or port number has changed in the license file, the p4 license command will not work. For example, if the IP address/MAC address changed in the new license file, you will receive the following message: "**`Server license IPaddress changed, cannot proceed.`**" In this case, stop and restart the server as detailed above.
>
> If p4 info does not indicate a license file update, stop and restart the server, then check the log file.

> **Tip**
> We recommend that you hide sensitive information from unauthorized users of `p4 info` by setting the dm.info.hide configurable.

Helix Visual Client (P4V) Administration tool

If the IP address or port number has not changed,

1. Launch P4V as a Helix server superuser.
2. Choose **Tools** > **Administration**.

3. On the Administration Home page, click **Load new license file** and browse to the license file on your local disk.
4. After successful installation, verify that the Administration tool Home page is updated with the new license information.

> **Tip**
> For more information, see the Support Knowledgebase article, "Moving a Helix Core Server".

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`. For more information, see *Using Helix Core Server for Distributed Versioning*.

Defining filetypes with p4 typemap

Helix server 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, Helix server 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 Helix server as being of type `text`.

The `p4 typemap` command solves this problem by enabling system administrators to set up a table that links Helix server 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 Helix server 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 Helix server 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 Helix server file types and modifiers for common file extensions.

File type	Helix server 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
`.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 Helix server 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 *Helix Core P4 Command Reference*.

Implementing site-wide exclusive locking with p4 typemap

By default, Helix server supports concurrent development. However, 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 file your users add to the depot after you update your typemap automatically has the `+l` modifier applied
- at any given time, only one user can have the file opened

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.

For more informatino, see "Working with depots" on page 109.

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 Helix 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 Helix 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 Helix 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 *Helix Core P4 Command Reference*.

Requiring a minimum client version

Helix server offers a mechanism to control which versions of client applications are able to connect to it.

To require a minimum version, set the configurable `minClient` to the appropriate revision.

You can also set the `minClientMessage` configurable to the error message that you want to be displayed when users attempt to connect to the server from a client application that needs to be upgraded.

For example:

```
$ p4 configure set minClient=2019.1
$ p4 configure set minClientMessage="Please upgrade to 2019.1 or
higher"
```

Rejecting client connection requests

By default, **all** clients can access the server, but you can block one or more client programs from accessing the Helix server by setting the `rejectList` configurable. Changes to this configurable take effect immediately, so no server restart is required.

> **Note**
>
> - To learn how to verify the blocking, see the Knowledge Base article, "How to Use the 'rejectList' Feature to Reject Client Connections".
> - The log does not include information about the rejected connection attempt.

The syntax for setting `rejectList`:

```
rejectList = programName [[,programName]...]
```

The syntax of `programName`:

```
programName[,version=versionName]
```

> **Tip**
>
> No wildcard character is allowed in the `programName`.

To block requests from **all** command line clients, regardless of the version:

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

Block specific versions

To block requests from versions you specify, use the default separator, which is the comma (`,`):

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

Block build number and platform

You can specify a version using a build number:

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

Or you can use platform information:

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

Or you can block for build number AND platform:

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

Tip
Use 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'
```

Blocking P4V clients from accessing Helix server

To block specific P4V versions, specify strings. For example, to block P4V clients version 2015.2 on Windows and Linux platforms, as well as version 2012.1 on Windows:

```
$ p4 configure set "rejectList=P4V/NTX64/2015.2, P4V/NTX86/2012.1,
P4V/LINUX26X86_64/2015.2"
```

Note
You can only use the `version=` field in the `rejectList` configurable for clients that specify their version in the `version` field. P4V passes its version information on the `program` string, not a `version` string. When you connect to Helix server with the command line client, the client specifies its `program name` as `p4` and its `version` as, for example, `2015.1/NTX64/1227227`. However, when you connect with P4V, P4V tells Helix server that its `program name` is `P4V/MACOSX106X86/2012.3/578478` and its `version` is `NULL`.

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 about system hardware and any non-default user preferences. 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 users from seeing the prompt. However, this is an incomplete solution because if users connect 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

For current releases of the server:

- The Helix Core server on UNIX supports case-sensitive names.
- The Helix Core 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 Helix Core server runs on, use `p4 info`.

Helix server on Linux

If your Helix Core server is on Linux, and you have users on both Linux and Windows, your Linux users must be careful not to submit files whose names differ only by case. Although the Linux server can support these files, when Windows users sync their workspaces, some files might overwrite 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 Linux user's workspace.

The Linux Helix server always respects case in client names, label names, branch view names, and so on. Windows users connecting to a Linux 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`, this client workspace is named `rocket` by default. If the user later sets `P4CLIENT` to `ROCKET` (or `Rocket`), the Helix server will display a message that the workspace is undefined. The user must set `P4CLIENT` to `rocket` (or unset it) to use the client workspace defined.

Helix server on Windows

If your Helix Core server is running on Windows, your UNIX users must be aware that it 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 Helix 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 Helix Core 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
- Helix server 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 Helix server `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 Helix 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, Helix server 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

> **Important**
> A Helix server that is in unicode mode cannot be changed to non-unicode mode.

- "Configuring a new server for Unicode" below
- "Configuring an existing server for Unicode" below, which requires a server restart

> **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 this 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:

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

This command verifies that all existing metadata is valid UTF8, and 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.

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 "Backup and recovery" on page 175.
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 which 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, informational and error messages are in English. You can localize Helix server messages. To ensure best results, contact Perforce Technical Support. The following overview explains the localization process.

To localize Helix 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 Helix server 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 proxy can also use this `db.message` file. See "Localizing P4P" on page 488.
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 90 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)" below or more information.
- You need to set **P4CHARSET** to an **utf16** or **utf32** setting.

  See "Controlling translation of server output" on page 90 for more information.
- The file is checked out using Helix server client applications that handle Unicode environments in different ways.

  See "Using other Helix server client applications" on page 90 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 Helix 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 Helix server UTF-8 validation
	Yes		**`utf8-bom`**	
	No		**`utf8unchecked`**	
	Yes		**`utf8unchecked-bom`**	
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 Helix server does not translate text files when you sync or submit them. Helix server 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 Helix server client applications

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

- **P4V (Helix 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.
- **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.
- **P4GT** and **P4EXP**, the Helix Plugin for File Explorer, 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 Helix 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 Helix server 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" on page 206.

Logging errors

Use the `-L` flag to `p4d` or the environment variable `P4LOG` to specify the Helix server 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.

Helix server also supports trace flags used for debugging. See "Diagnostic flags for monitoring the server" on page 204 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 Helix server 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 207 for details.

Configuring P4V settings

Not every site (nor every user at every site) requires the full suite of functionality in P4V, the Helix Visual Client. By using the `p4 property` command, it is possible for a user with at least *admin* privileges to control which P4V features are available for a given site, group, or user. Properties relate to performance, features, and Helix Swarm integration. Performance- and feature-related properties set at the server level override local P4V settings. Some properties can only be set on the server side.

If you add or update a property while P4V is running, P4V requires a restart before the new value takes effect. P4V reads properties that control features once, at startup, from the Helix server to which the user connects. For performance-related properties, if a user connects to a new Helix server after P4V startup, P4V reloads the properties from the server most recently connected to.

For information about configuring settings locally in P4V, see the *P4V User Guide*.

This section provides information about:

For more information on the **`p4 property`** command, see `p4 property` in the *Helix Core P4 Command Reference*.

Viewing effective P4V properties

To list P4V properties from the command line, run the **`p4 property`** command, as follows:

```
p4 property -l -n P4V.Features       // List enabled/disabled features
p4 property -l -n P4V.Performance    // List performance-related
settings
```

If no properties are listed, the user's local P4V preferences take effect.

Precedence of P4V settings

P4V settings take precedence based on how they were set and, if set on multiple levels, the sequence number.

Settings added using the **`p4 property`** command have the highest precedence. They override:

- Any central settings that may have been set using a P4JSAPI **`centralsettings.js`** file (for more information, see Administering P4V Settings Centrally in the *Javascript API for Visual Tools User Guide*)
- Any settings configured locally in the P4V user interface

If a system-wide value is set and other values exist for the same property, such as for individual users and one or more groups, the precedence depends on the sequence number for the property.

For example, the following output shows that the **`P4V.Features.Integration`** property is set system-wide, but also on a user and group level. User **`bill`** is a member of the **`p4users`** and **`dev`** groups. Which settings apply?

```
$ p4 property -l -A -n P4V.Features.Integration
P4V.Features.Integration = Off
P4V.Features.Integration = On (user bill)
```

```
P4V.Features.Integration = Off (group p4users)
P4V.Features.Integration = On (group dev)
```

If all versions of **`P4V.Features.Integration`** were created using the same sequence number, the answer would be:

1. System-wide, which takes precedence over
2. User, which takes precedence over
3. Group

However, if the sequence number is set to anything above 1, the highest sequence number wins. This means that if **`P4V.Features.Integration`** for user **`bill`** was created with a sequence number of 500 and the other versions have the default sequence number (1), the setting for **`bill`** takes precedence.

To view the sequence number for a property, an administrator can use the **`-ztag`** flag. For example:

```
$ p4 -ztag property -l -A -n P4V.Features.Integration
... name P4V.Features.Integration
... sequence 500
... value On
... time 1363106274
... modified 2013/03/12 16:37:54
... modifiedBy swood
... appliesToType user
... appliesTo bill
... name P4V.Features.Integration
... sequence 1
... value Off
... time 1363105851
... modified 2013/03/12 16:30:51
... modifiedBy swood
... name P4V.Features.Integration
... sequence 1
... value On
... time 1363102022
... modified 2013/03/12 15:27:02
... modifiedBy swood
... appliesToType group
... appliesTo dev
... name P4V.Features.Integration
... sequence 1
... value Off
... time 1363102040
... modified 2013/03/12 15:27:20
... modifiedBy swood
... appliesToType group
... appliesTo p4users
```

Performance-related P4V 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	P4V > Edit > Preferences	Default	Meaning	Version Introduced
`P4V.Performance.DirFetchSize`	Number of files per directory fetched at a time in the Depot tree	`0`, which means all files	Number of files to fetch at any one time. The minimum is `500` and we recommend `1000`.	2020.2
`P4V.Performance.FetchCount`	Number of changelists, jobs... to fetch at a time	`1000`	Number of changelists, jobs, branch mappings, or labels to fetch at any one time.	2013.1

Property	P4V > Edit > Preferences	Default	Meaning	Version Introduced
`P4V.Performance.OpenedLimit`	N/A	`1000`	Limits the number of files to check in the 'opened' call during a rollback operation. 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.	2013.1
`P4V.Performance.MaxFiles`	Maximum number of files displayed per changelist	`500`	Maximum number of files displayed per changelist.	2013.1
`P4V.Performance.MaxPreviewSize`	Maximum size of files to preview	`100`	Maximum size of files to preview, in kilobytes.	2013.1
`P4V.Performance.ServerRefresh`	Check server for updates every *X* minutes	`5`	Number of time between display refreshes, in minutes.	2013.1

Property	P4V > Edit > Preferences	Default	Meaning	Version Introduced
`P4V.Performance.AllowFullIstats`	Show pending stream-to-stream merge and copy hints	`On`	Enables/Disables global `istat` commands in P4V Stream graph. Users can still single-select a stream in the graph and refresh the stream to run the `istat` command for that stream and get the copy/merge flow information.	2013.4

Feature-related P4V properties

You can use the following properties to enable or disable features. These properties are read once, upon P4V startup, from the first service to which the user connects. Features that are deactivated by setting a property to `Off` are unavailable in P4V. Such features might still display in the P4V **Preferences** dialog, but you cannot override the configuration on the server side.

The following server properties can change the value of a feature that is `On` by default to `Off`. However, if a P4V user has set a feature to `Off` in the local preference, the local setting overrides the server property. For example, consider `P4V.Features.Streams`, which is `On` by default:

- If the user has `P4V.Features.Streams` set to `On` in the local preference, the Admin can force it `Off` by setting the server property to `Off`
- If the user has `P4V.Features.Streams` set to `Off` in the local preference, the Admin cannot force it `On` by setting the server property to `On`

Property	P4V > Edit > Preferences	Default Value	Description	Version Introduced
`P4V.Features.Administration`	Administration Tool	`On`	If `Off`, the Administration menu option is not displayed.	2013.1
`P4V.Features.CheckForUpdates`	Automatically check for Helix P4V updates.	`On`	If `Off`, disables the **Check for Updates** menu option on the **Help** menu. See also `P4V.Features.MaxAllowedVersion`.	2013.2
`P4V.Features.ConnectionWizard`	Set Up Connection Wizard	`On`	If `Off`, P4V does not attempt to use the **New Connection Wizard**.	2013.1
`P4V.Features.CustomTools`	Custom Tools	`On`	If `Off`, the **Manage Custom Tools** dialog is disabled.	2013.1
`P4V.Features.DashBoard`	N/A	`On`	If `Off`, the Dashboard is not displayed.	2013.1
`P4V.Features.DashBoard.Limit`	N/A	not set	If set, specifies the file limit for the Dashboard's Workspace Folder view. This setting overrides the the user-specified data preference on the server side.	2017.2
`P4V.Features.Dvcs`	DVCS - Distributed Version Control	`On`	`On` by default, but governed by the `server.allowpush` and `server.allowfetch` configurables on the shared server. Admins can disable the DVCS fature with this property. See also "Enabling distributed versioning" on page 76.	2017.1

Property	P4V > Edit > Preferences	Default Value	Description	Version Introduced
`P4V.Features.FullUserNames`	N/A	`Off`	If `On`, P4V displays full user names in the **Pending**, **Submitted**, and **History** tabs.	2018.4
`P4V.Features.HTMLTools`	HTML Tools	`On`	If `Off`, the HTML Tool editors are not enabled and HTML windows and tabs are not shown for this connection.	2019.2
`P4V.Features.Integration`	Merge, Copy and Branch Dialogs	`On`	If `Off`, users cannot integrate.	2013.1
`P4V.Features.Jobs`	Jobs	`On`	If `Off`, jobs support is disabled. Jobs do not appear in changelists, etc.	2013.1
`P4V.Features.Labeling`	Labels	`On`	If `Off`, the labels tab does not appear.	2013.1
`P4V.Features.Markdown`	Description fields	`On`	If `Off`, the Markdown feature does not appear.	2021.3

Property	P4V > Edit > Preferences	Default Value	Description	Version Introduced
`P4V .Features.MaxAllowed Version`	N/A		`<int value>` that determines the maximum version hint when checking for updates. Setting the version does not restrict the user from using a newer version of P4V against the server, but the checking for an update will not report a newer version than the one set in the hint. The value should be a changelist number equal to the maximum allowed version. If that value is set to 60000 and the latest current version is 65000, the user will get the message that there is no available update if the user is already at version 60000. If the user is at 60000 and the current live version is 70000, but the admin has set the max value to 65000, then the user will get a message that there is a newer version available, but an administrator has set a different maximum, and they should contact their administrator for the proper version.	2013.2

Property	P4V > Edit > Preferences	Default Value	Description	Version Introduced
`P4V.Features.PromptWorkspaceName`	Prompt for name when creating new workspace	`Off`	If `On`, P4V prompts the user for the workspace name when creating a new workspace. P4V runs the `p4 client -o` command, allowing a form-out trigger to modify the default form. P4V supports overwrite of the following attributes: `Client:`, `Root:`, `View:`, `Owner:`, `Description:`, `Host:`, `AltRoots:`, `ChangeView:`, `Options:`, `SubmitOptions:`, `LineEnd:`, and `Type:`.	2018.3
`P4V.Features.Repos`	Repos	`On`	If `Off`, repo-related icons and menus do not appear.	2020.2
`P4V.Features.RevisionGraph`	Revision Graph	`On`	If `Off`, the Revision Graph is disabled.	2013.1
`P4V.Features.Streams`	Streams	`On`	If `Off`, streams-related icons, menus, and the Stream Graph do not appear.	2013.1
`P4V.Features.Timelapse`	Time-lapse	`On`	If `Off`, Time-Lapse View is disabled.	2013.1
`P4V.Features.UnloadReload`	Unload/Reload	`On`	If `Off`, users cannot unload/reload a workspace, label, or task stream.	2013.2

Property	P4V > Edit > Preferences	Default Value	Description	Version Introduced
`P4V.Features.UnshelveSideways`	N/A	`On`	If `On`, allows users to unshelve files into a different branch or stream than they were shelved from. Shelved files can be unshelved directly from their parent stream, a non-parent stream, or a different depot using branch mapping.	2013.1
`P4V .Features.Workspaces`	N/A	`On`	If `Off`, users cannot edit or display their own (or other users') workspaces.	2013.1

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`.

Miscellaneous P4V properties

You can use the following properties to set P4V properties not related to performance or features.

Property	P4V > Edit > Preferences	Default	Meaning	Version Introduced
`P4V .Behavior.ModifiedFileIcon`	Use a distinct file icon for modified files		If `ON`, P4V uses a distinct file icon for files that users edit after syncing into their workspace.	2018.3

Property	P4V > Edit > Preferences	Default	Meaning	Version Introduced
`P4.DataAnalyticsPrompt`	Contribute your anonymous usage data to help us improve our products.	`On`	P4V 2015.1 inaugurated an opt-in program for collecting user data about interaction with our software. During the installation of P4V, a dialog prompts the end-user to decide whether or not to join the program. If the user chooses not to join, Perforce gathers no information about how that end-user uses P4V. The admin can disable the prompt by setting a property on the server: `p4 property -a -n P4.DataAnalyticsPrompt -v Off`	2015.1

Property	P4V > Edit > Preferences	Default	Meaning	Version Introduced
`P4V.Help.URL`	N/A	`perforce/<version>/manuals/p4v/#p4v/`	As of P4V 2014.2, P4V launches a web browser to display general or context-sensitive help information. Admins can download those web pages (`p4vsuite_en-help.zip`) from the FTP site and stage them locally. Set this property to the root path of the staged help. For detailed steps, see "Staging P4V help files locally" on page 105.	2014.2

P4VJS deployment properties

You can use the following properties to manage the deployment of HTML Tools in P4VS.

Property	Meaning
`P4VJS.HTMLTabs`	A URL search path to the definition of custom HTML Tabs
`P4VJS.HTMLWindows`	A URL search path to the definition of custom HTML Windows
`P4VJS.HTMLActions`	A URL search path to the definition of custom HTML Actions
`P4VJS.HTMLAllowList`	Overrides the local useAllowList and allowList preferences

For more information, see Deploy custom HTML pages in *P4VJS Developer Guide*.

Swarm integration properties

Property	Meaning	Version Introduced
`P4.Swarm.URL`	Set to the URL for the Helix Swarm server to enable the P4V integration with.	2014.3
`P4.Swarm.URL.xxxx`	If multiple Swarm servers exist, specify multiple Swarm URLs. `xxxx` is the server ID for the desired server.	2015.1
`P4.Swarm.Timeout`	Set the timeout value for the P4V integration with Swarm. By default, this is 10 seconds.	2014.3

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 a "Centralized authorization server (P4AUTH)" on page 458 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.

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
```

Staging P4V help files locally

If the P4V host does not have internet access, P4V cannot access the help files by default. In this case, you can make them available from a locally staged location.

Prerequesites

For locally staged help to work, both P4V and the Helix server need to be running version 2014.2 or later.

Staging location types

The following types of staging locations are known to work; others may work if a standard URI is available:

- A file system local to the P4V client host (or locally accessible). This could be a share mapped to a local drive letter on Windows, or a remote Unix filesystem mounted locally. This shared file location is not cross platform because you can only specify one path (Unix/Mac/Windows).
- A UNC share accessible to the P4V client host. This is only applicable to Windows clients.
- A website accessible to the P4V client. This can be made cross platform provided every client platform has access to the website.

Procedure

To stage help files locally:

1. Download the help files (**`p4vsuite_en-help.zip`**) from the FTP server. The exact location of this file varies depending on the version of P4V. The generic path looks as follows:

   - For 2014.2, 2014.3, and 2015.1:

     `http://ftp.perforce.com/perforce/<`
     `version`**`>/doc/help/p4vsuite/p4vsuite_en-help.zip`**

   - For 2015.2 and later:

     `http://ftp.perforce.com/perforce/<`
     `version`**`>/doc/manuals/p4vsuite_en-help.zip`**

   where **<*version*>** takes on a format of ***rxx.x***, such as **`r14.2`** or **`r17.1`**.

2. Unzip **`p4vsuite_en-help.zip`** to the required staging location.

   The staging location must be accessible to the P4V client, either as a file path or a URI.

   Following are examples for each type of staging location:

   - Local P4V client file system on Windows: **`C:\p4vsuite_en-help`**
   - Local P4V client file system on Linux/Unix: **`/var/www/html/p4vsuite_en-help`**
   - UNC share: **`\\myserver\myshare\p4vsuite_en-help`**
   - Web server (if you type this URL into a browser, it should list the "perforce" folder that is a subfolder of p4vsuite_en-help): **`http://mywebserver/p4vsuite_en-help`**

3. On the Helix server, set the **`P4V.Help.URL`** property.

   Note
   The property name is case sensitive.

   Following are examples for setting **`P4V.Help.URL`** for each type of staging location:

   - When staging from a local file system on Windows (note the use of forward slashes (**/**) as path separator, not backward slashes (\) as expected on Windows):

     ```
     p4 property -a -n P4V.Help.URL -v C:/p4vsuite_en-help/
     ```

   - When staging from a local file system on Linux/Unix:

     ```
     p4 property -a -n P4V.Help.URL -v /var/www/html/p4vsuite_
     en-help/
     ```

   - When staging from a UNC share (note the use of forward slashes (**/**) as path separator, not backward slashes (\)):

     ```
     p4 property -a -n P4V.Help.URL -v
     file://myserver/myshare/p4vsuite_en-help/
     ```

- When staging from a Web server:

```
p4 property -a -n P4V.Help.URL -v
http://mywebserver/p4vsuite_en-help/
```

4. Start P4V and go to **Help > P4V Help** to test if accessing the files works.

Troubleshooting P4V properties

If P4V is not picking up the value or setting you expected, check the following:

- Get the user to send full output from **Help > System Info** in P4V.
- Ask the admin to send the output from the following commands:

```
p4 -ztag property -l -A -n P4V.Features
p4 -ztag property -l -A -n P4V.Performance
p4 groups -u <user>
```

> **Important**
> Property names are case sensitive, so `P4V.Features.Integration` and `P4V.Features.integration` are *not* the same thing.

Windows configuration parameter precedence

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

1. For Helix server applications or a Helix server (`p4d`), command-line flags have the highest precedence.
2. For a Helix 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.

Working with depots

All 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. This kind of depot is also referred to as a *classic* depot. In addition, Helix server creates a default depot of type `graph` named `repo`. A graph depot serves as a container for Git repos. To be able to store Git data in a graph depot, you need to license Helix4Git. For more information on graph depots, see the *Helix4Git Administrator Guide*.

You can also 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 Helix server 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.

> **Note**
> Server extensions are versioned in a special `extensions` depot. For more information, see *Helix Core Extensions Developer Guide*.

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 *Helix Core 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.

Helix servers can host multiple depots, and Helix server client applications can access files from multiple depots. These other depots can exist on the Helix server normally accessed by the Helix server client, or they can reside within other, *remote*, 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 Helix server, use the **`p4 depots`** command.

Deleting depots

To delete a depot, it must be empty.

1. Remove all the files in the depot. See the `p4 obliterate` command.
2. Specify the depot to delete: `p4 depot` **`-d`** ***`depotname`***

For **`local`** depots and `"Spec depot" on page 112`, **`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 remain on the remote server remain. (See "Remote depots and multi-server development" on page 115.)

Before you use **`p4 obliterate`**, and *especially* if you’re about to use it to obliterate all files in a depot, see the warnings in "Reclaiming disk space by obliterating files" on page 240.

In a multi-server environment, the "Unload depot" on page 115 might have different contents on each edge server. The commit server does not verify that the unload depot is empty on every edge server. Therefore, you must specify `p4 depot -d -f` to delete the unload depot from the commit server.

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 Helix server normally accessed by the user's Helix server application. When using local depots, a Helix server application communicates with the Helix 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 Helix 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 Helix 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 Helix server. If you are using the distributed versioning architecture (DVCS), the personal server uses a stream-type depot.

See also p4 stream in the *Helix Core P4 Command Reference* and the Streams chapter of *Helix Core Server User Guide*.

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 Helix server 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. See "Controlling which specs are versioned" on page 114.

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 Helix server 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 Helix 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.p4s
```

> **Note**
> Certain `formtype`s (for example, the `protect`, `triggers`, and `typemap` forms, plus license updates) might have several versions, but only have a single incidence of that type of form on the server. Such `formtype`s have no `objectname`. Referencing such `formtype`s in the `SpecMap` field of the spec depot requires the suffix `.p4s` or the wildcard `*`.

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 238.

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 268.

Remote depots and multi-server development

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

Partitioning joint development projects into separate Helix server installations does not improve throughput, and usually only complicates administration. If your organization has developers in multiple sites working on the same body of code, it is better to set up a multi-server installation.

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 Helix server specified by the user's P4PORT environment variable or equivalent setting as a means to access a second, *remote*, Helix server. The local Helix server communicates with the remote Helix server 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 Helix server of type **remote**. It acts as a pointer to a depot of type "local" that resides on a second Helix server.
- A user of a remote depot is typically a build engineer or handoff administrator responsible for integrating software between separate organizations.

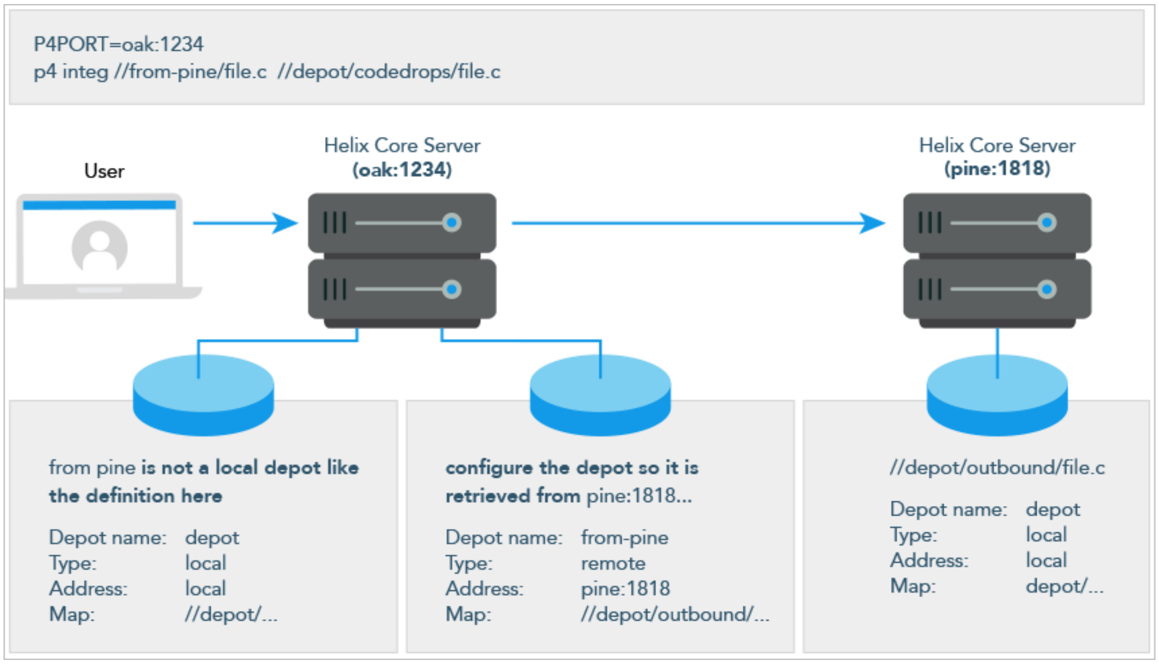
- Control over what files are available to a user of a remote depot resides with the administrator of the remote server, *not* the users of the local server.
- See "Restricting access to remote depots" on page 118 for security requirements.

For an alternative option to share code, see "Distributed development using Fetch and Push" on page 228.

How remote depots work

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

In this example, an administrator of a Helix 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
```

from pine **is not a local depot like the definition here**

Depot name:	depot
Type:	local
Address:	local
Map:	//depot/...

configure the depot so it is retrieved from pine:1818...

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

//depot/outbound/file.c

Depot name:	depot
Type:	local
Address:	local
Map:	depot/...

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 Helix server 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 Helix server 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 things must be configured:

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

  This is described in "Defining remote depots" below.

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

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

- 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 120.

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 Helix server to contact the remote Helix 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 "Configure workspace views" in the *Helix Core Server User Guide*.

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 **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 Helix 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 Helix 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**
> A Helix server at release 2010.2 authenticates as `remote` to an older Helix server and either as `remote` (if no service user is configured) or as the service user (if configured) to a Helix server at release 2010.2 and above.

By default, all files on a Helix 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 Helix server release 2010.2, it remains good practice to deny access to user **remote**. If the 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 the next page, 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 Helix server 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 232). 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 Helix 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/...
```

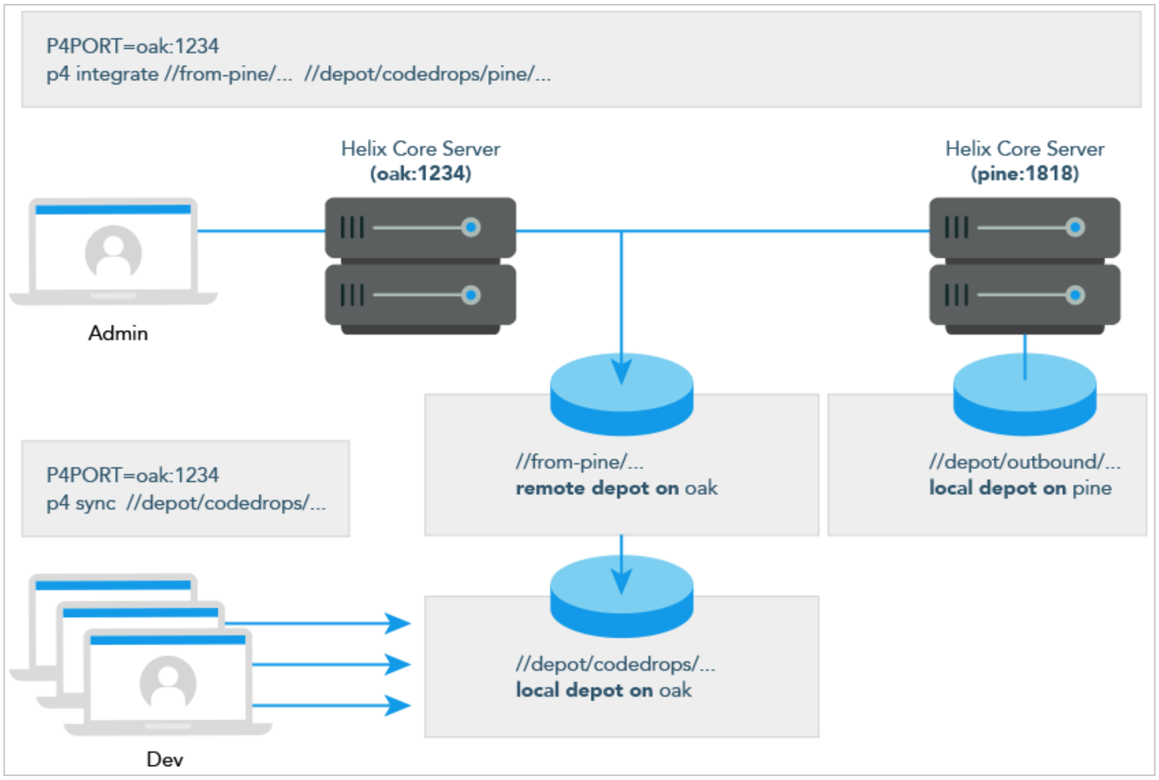
Only servers at IP address 192.168.41.2 that have valid tickets for the **pine** site's **service-oak** user, are permitted to access the **pine** server through remote depots, and only **//depot/outbound/...** is accessible.

Receiving a code drop

To perform a handoff or code drop between two Helix server 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 Helix server 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/...
```

```
P4PORT=oak:1234
p4 sync  //depot/codedrops/...
```

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 Helix server" on the next page 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 129.
- 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 server user database. See "Authentication options" on page 129 for information about how you can authenticate users.
- Access is defined using a protections that determine which Helix server commands can be run, on which files, by whom, and from which host. See "Authorizing access" on page 147 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 Core Server Administrator Guide*.

Before you can configure access and authentication, you must create users as described in "Managing users" on page 231.

Recommended settings to configurables for security

After installing Helix server, it is good practice to:

- hide sensitive information from unauthorized users of `p4 info` by setting the dm.info.hide configurable
- require ticket-based authentication by setting the `security` configurable to **3** or **4**
- prevent the automatic creation of new users by setting the `dm.user.noautocreate` configurable to **1** or **2**
- force new users that you create to reset their passwords by setting the `dm.user.resetpassword` configurable to **1**

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.

See also the Support Knowledgebase articles on "Securing Your Perforce Server" and "Connecting an ssh client to Perforce through a firewall".

Using SSL to encrypt connections to a Helix server

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

For any given Helix server, proxy, or broker, SSL encryption is an all-or-nothing option: If a Helix server is configured to use SSL (presumably for security reasons), all Helix server applications must be configured to use SSL. Conversely, if a Helix 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 Helix 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 127.

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 Helix Core client applications to explicitly specify the protocol to use when establishing a client-server connection.

plaintext	ssl
`tcp:`*`host`*`:`*`port`*	`ssl:`*`host`*`:`*`port`*

plaintext	ssl
`tcp:123.45.67.8:1666`	`ssl:123.45.67.8:1666`
	The first time that users connect to an SSL-enabled Helix Core Server, their client application informs them of the certificate fingerprint of the server’s identity key. If users can independently verify that the fingerprint is accurate, they should add the server to their `P4TRUST` file by: - using the `p4 trust` command, or - following the prompts in the client application, or - 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 Helix 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 Helix 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: Helix 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, that is, 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=Helix 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:F0:5F:31:6E:33:6F:0E: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 Helix server 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 Helix server 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, Helix server'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 Helix server, 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 Helix Broker may be configured to accept plaintext connections from older Helix server applications, and to forward those requests (encrypted by SSL) to a Helix server that requires SSL connections.
- A Helix 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 Helix server applications could use SSL to connect directly to the upgraded Helix server (by setting `P4PORT` to `ssl:new-server:1667`), while users of older Helix server applications could continue to use plaintext when connecting to a Helix 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 Helix Proxy and the Helix 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 Helix 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.

SSL and TLS Protocol Versions

By default, new clients connecting to new servers use TLSv1.2.

Clients and servers choose the highest TLS version supported by both ends of the connection.

If the "client" is not explicitly set, explicitly setting the server's ssl.tls.version.min and ssl.tls.version.max configurables will apply to "client" connections for backwards compatibility.

Configurables

Two server configurables restrict the allowed TLS versions when a new client connects to a new server:

`ssl.tls.version.min` [default=`10`]

`ssl.tls.version.max` [default=**`12`**]

Each of these configurables can take one of the following values:

`13` specifies TLSv1.3

`12` specifies TLSv1.2

`11` specifies TLSv1.1

`10` specifies TLSv1.0

- **`ssl.tls.version.min`** configurable specifies the lowest TLS version that will be accepted
- **`ssl.tls.version.max`** specifies the highest TLS version that will be accepted.

Note
The 2021.1 release introduced two corresponding configurables on the client side:

ssl.client.tls.version.min

ssl.client.tls.version.max

The range of values that the client-side configurables accept is not necessarily identical with the range of values that the server-side configurables accept. Clients and servers choose the highest TLS version supported by both ends of the connection.

Important
These "client" and server) of configurables can be used in servers, proxies, or brokers where both upstream (client-side) and downstream (server-server) connections are made. For example, in edge-to-edge chaining, one edge server acts as a "client" to another edge server. This aspect of a server as a "client" applies to other scenarios as well, such as centralized authorization server (P4AUTH), centralized changelist server (P4CHANGE), and when one server accesses a "remote depot" on another server.

Important
After you change the value of these configurables, you must explicitly "stop" the server.

`p4 admin restart` is NOT sufficient.

The change takes effect after a complete "stop" and start.

- For UNIX, see "Stopping the Helix server" on page 48 and "Starting the Helix server" on page 48.
- For Windows, see "Starting and stopping the Perforce service" on page 51.

Tip
TLS 1.3 is faster than TLS 1.2 at file transfers, but establishing a TLS 1.3 connection requires more overhead. (Note that the higher the latency, the less the connection overhead matters.)

> Applications that rely on many short-lived connections might want to pin their version to 1.2 if using a 1.3-enabled server.

If the client-side configurables are not set

To force the use of TLSv1.3, set

```
ssl.tls.version.min=13
ssl.tls.version.max=13
```

To force the use of TLSv1.2, set

```
ssl.tls.version.min=12
ssl.tls.version.max=12
```

To allow TLSv1.2 or TLSv1.3, but exclude TLSv1.0 and TLSv1.1, set

```
ssl.tls.version.min=12
ssl.tls.version.max=13
```

These configurables can also be used by clients for testing purposes or to prevent connecting to servers below a minimum version.

On a client, to verify that TLSv1.0 does not connect:

```
p4 -v ssl.tls.version.min=10 -v ssl.tls.version.max=10 info
```

Values of either configurable outside of the legal range will be treated as if they were pinned to the nearest end of the range. Thus values below `10` will be treated as `10`, and values above `13` will be treated as `13`.

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. See the Support Knowledgebase article "Connecting an ssh client to Perforce through a firewall".

Authentication options

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

Overview

User authentication can take place using any of the following options:

- Using "Helix Authentication Service" on page 134 in conjunction with an identity provider (IdP)
- "LDAP authentication" on page 134 against an Active Directory or LDAP server that is accessed according to an LDAP specification. Enabling this option disables trigger-based authentication.
- Against Helix server’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 141.
- 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 329.

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 Helix server’s internal user database. For more information, see "Defining authentication for users" on page 133.

When logging in using either authentication method, Helix server 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. Helix server 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=securitylevel
```

where *`securitylevel`* is `0`, `1`, `2`, `3`, `4`, `5`, or `6`:

Security level	Server behavior
`0` (or unset)	The **default security level** `0` does not require passwords and does not enforce password strength. **Warning** We strongly recommend that when you create a new user, you assign that user an initial password, and that you make it a strong password. A new user with no password can run p4 passwd unchallenged. For example, `p4 -u newUser passwd` allows anyone who knows the value of `newUser` to set the new password without any prior authentication. This security issue is present even though security levels higher than level `1` require passwords for all user accounts. 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 Helix server 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. See "Password strength requirements" on page 142. Very old Helix server 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 Helix server 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.

Security level	Server behavior
`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.
`5`	Requires that any intermediary (such as a proxy or broker) has a valid authenticated service user.
`6`	Requires each intermediary to have a valid server spec, where the service user must match the user named in the `User` field of the spec. The server spec is found by matching the intermediary's P4PORT with a value in the `AllowedAddresses` field of the spec. For example, if connecting to a proxy on `10.0.0.100:1667`, a server spec with this IP address and port number in the `AllowedAddresses` field must exist and must specify the proxy's service user in the `User` field. Errors relating to configuration of intermediaries are logged to the `route.csv` logfile, if structured logging is enabled. See "Enable and configure structured logging" on page 208.

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

Authentication triggers or LDAP

Important
When user authentication occurs through authentication triggers or the native LDAP configuration, if `security` is:

- unset, or set to `0`, `1`, or `2`, the server behaves as if the security level is set to `3`
- set to `3` or higher, the server uses that setting

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 Helix server'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`.

> **Warning**
> By default, Helix server creates a new user whenever a previously unknown user invokes any command that can update the repository or its metadata. When executed by a nonexistent user, most Perforce commands cause a user to be created. You can control this behavior by setting the `dm.user.noautocreate` configurable with the `p4 configure` command. For greatest security, we recommend that only the Helix server superuser be allowed to create new users:
>
> ```
> $ p4 configure set dm.user.noautocreate=2
> ```

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 Helix server, you can use the group-related fields of the LDAP configuration to implement a basic authorization step that filters out non-Helix server 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" section of the *Helix Core P4 Command Reference*.

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

Helix Authentication Service

Helix Authentication Service (HAS) enables you to integrate Helix Core or Helix ALM with your organization's Identity Provider (IdP), such as AuthO, Azur Active Directory, Okta (identity management), OneLogin, Google G Suite IdP, Ping Identity, Cisco Duo Security, or others.

HAS supports Security Assertion Markup Language (SAML) and OpenID Connect (OIDC).

For more information about HAS, see *Helix Authentication Service Administrator Guide*.

LDAP authentication

The following sections explain how you can authenticate against Active Directory and LDAP servers.

Authenticating against Active Directory and LDAP servers

LDAP, Lightweight Directory Access Protocol, is supported by many directory services, including Active Directory and OpenLDAP. Helix server offers two ways of authenticating against Active Directory or LDAP servers: using an authentication trigger or using an LDAP specification. We recommend using an LDAP specification because it:

- is easier to use
- requires no external scripts
- allows users who are not in the LDAP directory to be authenticated against the internal user database
- 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 but SAML is not.

The steps required to set up configuration-based LDAP authentication are described in the following sections. Information relating to LDAP authentication applies equally to using Active Directory.

Overview of the configuration process:

- 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 Helix 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 "Server security levels" on page 130 `3`.

Creating an LDAP configuration

An *LDAP configuration* specifies an Active Directory or other LDAP server against which the Helix 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:                   UK_LDAP
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 Helix 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 Helix 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:  CN=bruno, DC=foo, DC=com
  SearchPasswd:  ******
  ```

  ```
  BindMethod:    search
  SearchBaseDN:  ou=users,dc=example,dc=com
  SearchFilter:  (&(objectClass=user) (sAMAccountName=%user%))
  SearchScope:   subtree
  SearchBindDN:  CN=bruno, DC=foo, DC=com
  SearchPasswd:  ******
  ```

See the *Helix Core P4 Command Reference* for information about the `p4 ldap` command and the LDAP specification. The LDAP spec also allows you to control the behavior of LDAP integration:

- 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. This 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=UK_LDAP
```

(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" below.

If you are using multiple directory servers for "Failover" on page 191 or user management, you might need to create multiple LDAP configurations. In this case,

- create an LDAP configuration for each LDAP server
- 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 cannot have two LDAP configurations using the same order number for the same Helix 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 Helix server or using LDAP.
  - If **`auth.default.method=perforce`** and you want only the Helix server superuser to create new users, set `dm.user.noautocreate` to **`2`** explicitly.
  - If **`auth.default.method=ldap`**, **`dm.user.noautocreate`** is **`2`** implicitly.
- `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.
- `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.pagesize` - helps you manage LDAP searches with paged results by setting limits to page size.

For example, the following commands 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=UK_LDAP
$ p4 configure set auth.ldap.order.2=US_LDAP
$ p4 configure set auth.ldap.order.5=RU_LDAP
$ p4 configure set auth.default.method=perforce
```

For additional information about authentication-related configurables, see the Configurables in the *Helix Core 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 Helix 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 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 a Helix server, 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
```

Important
LDAP queries for a user are performed as that user. Therefore, a user must be a member of a group before that user can see that group.

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 log in if 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 olivia
   ```

   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`** configurable to tell Helix server in what order to use this configuration. For example:

   ```
   $ p4 configure set auth.ldap.order.1=bruno
   ```

   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
   ```

   > **Note**
   > This disables authentication trigger support.

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 `p4 ldaps -t` command in the *Helix Core 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 *Helix Core 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 329. For information about group authorization, see the next section.

Authenticating using passwords and tickets

Helix server 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 Helix server.
>
> To encrypt network traffic between client workstations and the Helix server, configure your installation to use SSL. See "Using SSL to encrypt connections to a Helix server" on page 123.

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 130 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 Helix server-based and LDAP-based authentication are enabled.)

Password strength requirements

Certain combinations of security level and Helix server applications releases require users to set "strong" passwords. Helix Core server defines a strong password as:

- at least `dm.password.minlength` long, which, by default, is `8` characters
- contains at least two of the following :
  - Uppercase letter(s)
  - Lowercase letter(s)
  - Non-alphabetic character(s)

Although `abcd1234` is by default, considered a strong password in an environment with the security configurable set to `2`, it is too easy to guess.

> **Tip**
> To create secure password that is easy-to-remember:
>
> 1. Start with a phrase, such as `Perforce Enterprise-class Version Control.`
> 2. Make the phrase resemble a single word, such as `PEnterprise-classVC.`

> 3. Represent some letters with non-alphabetical characters:
>   `PN2prI$-k|@zV(.`

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

Helix server superusers can manually set a 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 Helix server, 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 Helix server. Helix server creates a ticket for a user when they log in using the `p4 login -a` command. Helix server 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, a Helix server accepts valid tickets wherever users can specify Helix server 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`.

A ticket expires:

- If the user's AuthMethod is changed
- If the user's password is changed and the user is using `AuthMethod` of `perforce`.
- When the ticket's password expires. This assumes that password aging is in effect.

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 Helix server, 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 Helix server 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, Helix server 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 Helix server 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 Helix server

To log out of Helix server 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 Helix serverbut 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 Helix server from more than one machine, you can log out from all machines from which you were logged in by using the command:

```
$ p4 logout -a
```

All of your Helix server 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
```

Multi-factor authentication

Most Helix Core servers are behind a secure firewall and require user passwords.

MFA in general

Multi-factor authentication (MFA) adds an additional layer of security in case a user password is compromised. MFAis a method of confirming a user's claimed identity. A user is granted access only after successfully presenting two or more pieces of evidence (or factors) to an authentication mechanism, such as:

- knowledge (something they and only they know)
- possession (something they and only they have)
- inheritance (something they and only they are)

MFA with Helix Authentication Service

If you are using the Helix Authentication Service (HAS) and you want multi-factor authentication, it is strongly recommended that you use the MFA solution that your IdP provides. This means you should not install the separate Helix MFA app. The only use case for installing the Helix MFA app with the Helix Authentication Service is if you wanted to use a MFA service that is separate from your IdP. For information about HAS, see *Helix Authentication Service Administrator Guide*.

Helix MFA app

Helix MFA app:

- should only be used when your password store and your MFA service are separated. A common example would be using LDAP as your password store with Okta as your MFA service.
- supports the most common factors:
  - One Time Password (OTP) codes
  - Third party or external prompts, such as a mobile app authentication or a phone call

For an example of how the Helix Core server can support MFA in conjunction with a cloud-based identity provider, see:

- the Perforce Okta MFA trigger at https://swarm.workshop.perforce.com/projects/perforce_software-mfa/files/main/okta/okta-mfa.rb
- "Triggering for multi-factor authentication (MFA)" on page 340, which:
  - explains the three types of triggers necessary for Helix MFA (`auth-pre-2fa`, `auth-init-2fa`, and `auth-check-2fa`)
  - shows an example of an `auth-check-2fa` trigger that Perforce has validated with Okta. To find out more about Okta and the factors it supports, contact your Okta administrator or see https://support.okta.com/help
  - includes comments intended to make this example a starting point for working with the API of other services that support MFA

Authorizing access

Helix server provides a protection scheme to prevent unauthorized or inadvertent access to files in the depot. The protections determine which Helix server 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 Helix server for the first time. Before the first call to **`p4 protect`**, every Helix server 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 Helix server 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 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.)

Proxy and protections

To apply the IP address of a Helix Proxy user’s workstation against the protections table, prepend the string **`proxy-`** to the workstation’s IP address.

Important
Before you prepend the string **`proxy-`** to the workstation’s IP address, make sure that a broker or proxy is in place.

For instance, consider an organization with a remote development site with workstations on a subnet of `192.168.10.0/24`. The organization also has a central office where local development takes place; the central office exists on the `10.0.0.0/8` subnet. A Perforce service resides in the `10.0.0.0/8` subnet, and a Helix Proxy resides in the `192.168.10.0/24` subnet. Users at the remote site belong to the group `remotedev`, and occasionally visit the central office. Each subnet also has a corresponding set of IPv6 addresses.

To ensure that members of the `remotedev` group use the proxy while working at the remote site, but do not use the proxy when visiting the local site, add the following lines to your protections table:

```
list    group    remotedev     192.168.10.0/24               -//...
list    group    remotedev     [2001:db8:16:81::]/48         -//...

write   group    remotedev     proxy-192.168.10.0/24          //...
write   group    remotedev     proxy-[2001:db8:16:81::]/48    //...

list    group    remotedev     proxy-10.0.0.0/8              -//...
list    group    remotedev     proxy-[2001:db8:1008::]/32    -//...

write   group    remotedev     10.0.0.0/8                     //...
write   group    remotedev     [2001:db8:1008::]/32           //...
```

The first line denies `list` access to all users in the `remotedev` group if they attempt to access Helix server without using the proxy from their workstations in the `192.168.10.0/24` subnet. The second line denies access in identical fashion when access is attempted from the IPV6 `[2001:db8:16:81::]/48` subnet.

The third line grants `write` access to all users in the `remotedev` group if they are using a Helix Proxy server and are working from the `192.168.10.0/24` subnet. Users of workstations at the remote site must use the proxy. (The proxy server itself does not have to be in this subnet, for example, it could be at `192.168.20.0`.) The fourth line grants access in identical fashion when access is attempted from the IPV6 `[2001:db8:16:81::]/48` subnet.

Similarly, the fifth and sixth lines deny `list` access to `remotedev` users when they attempt to use the proxy from workstations on the central office's subnets (`10.0.0.0/8` and `[2001:db8:1008::]/32`). The seventh and eighth lines grant write access to `remotedev` users who access the Helix server directly from workstations on the central office's subnets. When visiting the local site, users from the `remotedev` group must access the Helix server directly.

When the Perforce service evaluates protections table entries, the `dm.proxy.protects` configurable is also evaluated.

`dm.proxy.protects` defaults to `1`, which causes the `proxy-` prefix to be prepended to all client host addresses that connect via an intermediary (proxy, broker, replica, or edge server), indicating that the connection is not direct.

Setting `dm.proxy.protects` to `0` removes the `proxy-` prefix and allows you to write a single set of protection entries that apply both to directly-connected clients as well as to those that connect via an intermediary. This is more convenient but less secure if it matters that a connection is made using an intermediary. If you use this setting, all intermediaries must be at release 2012.1 or higher.

Permission subfields

Each line specifies values for the subfields:

Subfield	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`). ■ 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 Helix server via the Helix Proxy, see "Helix Proxy" on page 482.

Subfield	Meaning
`Files`	A file specification representing the files in the depot on which permissions are being granted. Helix server 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 Helix server commands that display file metadata, such as `p4 filelog`. No permission is granted to view or change the contents of the files.
`read`	The user can run those Helix server 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 Helix server 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`**.

Level	Meaning
`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 155 for details.
`admin`	For Helix server administrators; grants permission to run Helix server 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 Helix server superusers; grants permission to run all Helix server 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 Helix server 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 Helix server command, see "How protections are implemented" on page 162.

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 might already have open. He can continue to create and modify files, but only if those files are outside of the protected **`//depot/build/...`** area.

Stream spec permissions

`readstreamspec`	The user can display a stream spec with `p4 stream -o`
`=readstreamspec`	If this right is denied, users cannot execute `p4 stream -o`
`openstreamspec`	This gives the user permission to revert, resolve, shelve, or open for edit a stream spec.
`=openstreamspec`	If this right is denied, users cannot revert, resolve, shelve, or open for edit a stream spec.
`writestreamspec`	The user can submit or modify a stream spec.
`=writestreamspec`	If this right is denied, users cannot submit or modify a stream spec.

Note
If any streamspec permissions exist for any user:

- users without explicit streamspec permissions have no access to stream specs
- `list` continues to provide p4 streams access

If no streamspec permission exists for any user:

- `list`, `open`, and `write` permissions control stream spec access for the p4 edit, p4 resolve, p4 revert, p4 shelve, p4 submit, p4 streams, and p4 stream commands.
- `list` grants `p4 streams` access for stream spec paths.
- Any `open` or higher permssion for a user anywhere grants stream spec `edit` and `write` permissions to that user for all stream specs.

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 Helix server 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

Lisa, whose Helix server 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 because on 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 can 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 can 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, 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 162 for a more detailed explanation.
- Without exclusionary mappings, the order of items in the protections table is not important.

Example

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 Helix server 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.

Helix server 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 *Helix Core P4 Command Reference* for information on how to create a spec depot.

Granting access to groups of users

Helix server *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 Helix server superusers can use these commands. (Helix server 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 Helix server users and groups with LDAP groups" on the facing page.

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. To create users, use the **`p4 user`** command.

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 **Granting access to Helix 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 Helix server users and groups with LDAP groups

You can configure Helix server to automatically synchronize the contents of a given Helix server user or user group with that of an LDAP user or group. Protections are still assigned based on the identity of the Helix server user or group (using the `p4 protect` command), but which users are included in the Helix server 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 *Helix Core P4 Command Reference*.

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 Helix server`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
     - 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 Helix server group.
2. Define a group owner for the Helix server group. The owner does not have to be a member of the corresponding LDAP group.
3. Use the `p4 ldapsync` command, specifying which Helix server 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, make the `p4 ldapsync` command run at startup time and specify the value of the interval. See the Examples in the p4 ldapsync command in *Helix Core P4 Command Reference*.

The following examples, included in "Synchronizing with Active Directory" on the facing page and "Synchronizing with OpenLDAP" on the facing page, 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 Helix server 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, 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 Helix server 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=agupta, OU=users, DC=foodomain, 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=agupta, OU=users, DC=foodomain, 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.

How protections are implemented

This topic assumes you have read p4 protect in *Helix Core P4 Command Reference*.

For any given command by any user, the server reads the protections table for a line matching user or group, IP address, file path, and access level.

If more than one line matches, the last line "wins".

The table contains the following columns:

- access level, such as **`write`** or **`super`**
- **`group`** or **`user`** - to keep the table short, we recommend combining users into groups
- name of that **`group`** or **`user`** - although **`*`** is allowed, we recommend using specific groups
- IP Address for the server (host), or **`*`** for any IP Address
- Comments, which have the **`##`** prefix

A protections table might look like this:

```
write   group   Dev1  *                       //depot/...
list    group   Dev1  *                       -//depot/proj/...   ##
exclusionary mapping
write   user    Maria 192.168.100.0/24        //...
super   user    Alice *                       //...   ## put super user
at the bottom
```

Suppose that user **`Maria`** is a member of **`Dev1`**, the organization is using IPv4 connections, and Maria attempts four operations:

From IP address...	Maria tries...	Result
`10.14.10.1`	`p4 print` **`//depot/misc/...`**	Succeeds because **`Dev1`** can write anywhere in the depot except the **`proj`** area.

From IP address...	Maria tries...	Result
`10.14.10.1`	`p4 print //depot/proj/README`	Fails because the the exclusionary mapping of `-//depot/proj/...` removes all of Maria’s permissions on any files in this directory.
`192.168.100.1`	`p4 print //depot/proj/README`	Succeeds because at this IP address, Maria's **`write`** permission implies the ability to print. The last line that matches "wins".
`192.168.100.1`	`p4 verify //depot/misc/...`	Fails because the p4 verify command requires **`super`** access. Maria only has **`write`** access.

Without an exclusionary mapping, the most permissive line rules

An exclusionary mapping begins with the minus sign (`-`). For example, `-//depot/dev/productA/...`

Suppose user Maria is a member of **`Dev1`** and **`Dev2`**, and the protections table contains no exclusionary mapping:

```
write        group        Dev2     *     //depot/dev/...
read         group        Dev1     *     //depot/dev/productA/...
write        group        Dev1     *     //depot/elm_proj/...
```

Maria can run **`p4 edit //depot/dev/productA/readme.txt`** because all members of the **`Dev2`** group can do so.

> **Note**
> If at least one group to which the user belongs can run the command, that user can run the command.

Exclusion overrides prior inclusion

Suppose the protections table contains an exclusionary mapping, `-//depot/dev/productA/...`, that prevents Maria from editing files in the **`productA`** directory:

```
write    group    Dev1    *    //depot/dev/...             ## Maria is a
member of Dev1
list     group    Dev1    *    -//depot/dev/productA/...   ## exclusionary
mapping overrides the line above
```

```
super   group   super-users   *   //...               ## Maria is NOT
a member
```

If Maria attempts to run **`p4 edit //depot/dev/productA/readme.txt`**, she cannot because she is excluded from **`/depot/dev/productA/...`** and no lower line overrides that exclusion.

Exclusionary mappings and admin

In this case,

```
write   group   Dev1   *   //depot/dev/...
write    group   Dev1   *   -//depot/dev/productA/...
super   group   super-users   *   //...
```

the members of Dev1 have no access to productA.

However, in this case,

```
admin   group  Admins  *   //depot/...
write   group  Admins   *   -//depot/dev/productA/...
super   group  super-users   *   //...
```

although the members of Admins cannot edit the files in productA, they can run administrative commands, such as p4 changes.

Consider whether any exclusionary mappings are present

If exclusionary mappings are NOT present	If exclusionary mappings ARE present
▪ A user is granted the highest permission level listed in the union of **all the mappings that match** the user, the user’s IP address, and the files the user is trying to access.	▪ A matching exclusionary mapping overrides any matching protections listed above it in the table.

`Exclusionary mapping and` =

`-` before the file path:

```
write   group   Dev1    *   //depot/dev/...
list   group    Rome    *   -//depot/dev/prodA/...    ## exclusion of list implies no read, no write, etc.
read    group   Rome    *   //depot/dev/prodA/...    ## Rome can only read this one path
write   group   Dev2    *   //depot/elm_proj/...
super   user    Anne    *   //...
```

`-` before the file path and `=` before the access level

```
write   group   Dev1    *   //depot/dev/...
=read   group   Rome    *  -//depot/dev/prodA/...    ## Rome cannot read this one path
write   group   Dev2    *   //depot/elm_proj/...
super   user    Anne    *   //...
```

See also the "Permission levels and access rights" in the description of p4 protect in the *Helix Core P4 Command Reference*.

Access levels and =

If the line is ...	... for the specified user, IP address, and file path ...
`read group Dev1 * -//depot/dev/prodA/...`	denies read and its lesser permissions
`=read group Dev1 * -//depot/dev/prodA/...`	denies read but allows its lesser permissions
`open group Dev1 * -//depot/dev/prodA/...`	denies open and its lesser permissions
`=open group Dev1 * -//depot/dev/prodA/...`	denies open but allows its lesser permissions
`write group Dev1 * -//depot/dev/prodA/...`	denies write and its lesser permissions
`=write group Dev1 * -//depot/dev/prodA/...`	denies write but allows its lesser permissions
Unlike `read`, `open`, and `write`, `=branch` always has the `=`	
`=branch group Dev1 * -//depot/dev/prodA/...`	denial of integrate to that path

Access levels required by Helix server 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. (See p4 protect)

Command	Access Level	Notes
`add`	`open`	
`admin`	`super`	An operator with `admin` can use all options except `updatespecdepot`, `resetpassword`, and `end-journal` A user with `super` can use all options
`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.
`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`	

Command	Access Level	Notes
`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`	
`edit`	`open`	
`export`	`super`	
`extension`	`super`	The super user can delegate some permissions to admins and users.
`fetch`	`admin`	

Command	Access Level	Notes
`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`	
`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.

Command	Access Level	Notes
`labelsync`	`open`	
`ldap`	`super`	
`ldaps`	`super`	
`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`	

Command	Access Level	Notes
`print`	`read`	
`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 Helix 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.

Command	Access Level	Notes
`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.
`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`	
`streamspec`	`admin`	
`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`	

Command	Access Level	Notes
`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`	
`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 Helix server superusers. See "Forcing operations with the -f flag" on page 226 for details.

For additional details, see p4 protect in *Helix Core P4 Command Reference*.

Restrict access to changelists

We recommend that you restrict access to changelists by issuing this command: `p4 configure set defaultChangeType=restricted`

By default, all users can view a pending or submitted changelist, regardless of whether they are permitted access to the files in the changelist by the protections table.

The visibility of restricted changelists:

- **Pending changelists**: Visible only to owner, regardless of whether other users have access to checked-out files.
- **Pending changelists containing shelved files**: Users with *list* (or higher) permission (as specified in the protection table) to one or more of the shelved files can view those files but cannot view the changelist description.
- **Submitted changelists**: Users with *list* (or higher) permission (as specified in the protection table) to one or more of the submitted files can list those files and read the changelist description.

Backup and recovery

This chapter describes the commands and processes you use to back up and recover your Helix Core server. For information about multi-server backup and recovery, see the "Backing up and upgrading services" on page 492 section under "Deployment architecture" on page 369.

Warning
To reduce the risk of data loss,

- Validate your backup procedures and follow them
- Take checkpoints and rotate journals on a regular basis.

Tip
If your site is very large (gigabytes of `db.*` files), creating a checkpoint might take a substantial amount of time. Therefore, consider:

- only creating a checkpoint at the end of each work week, while rotating the `journal` file during your nightly backup
- performing checkpoints on a separate instance of your Helix Core p4d Server database. See the Support Knowledgebase article, Offline checkpoints.

Version files versus database files (metadata)

The Perforce service stores two kinds of data: *versioned files* and *metadata*.

- *Versioned files* are files submitted by Helix server 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 Helix server 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.

Backup and recovery concepts

Disk space shortages, hardware failures, and system crashes can corrupt any Helix server files. That's why the entire Helix server root directory structure (your versioned files and your database) must be backed up regularly.

The versioned files are stored in subdirectories beneath your Helix server root and can be restored directly from backups without any loss of integrity.

The files that constitute the Helix server 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 Helix server 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 Helix server database.

> **Warning**
> Checkpoints and journals archive only the Helix server 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 Helix server database is as irreplaceable as your versioned files, checkpointing and journaling are an integral part of administering Helix server, 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 Helix server database. When you create a checkpoint, the 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 Helix 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 Helix 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:

```
$ p4d -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 Helix server 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, Request Support immediately. Checkpoint failure is usually a symptom of a resource problem 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 file and the journal file that was collected from then until Wednesday, those two files contain the same information as a checkpoint made Wednesday. If a disk crash were to cause corruption in your Helix server database on Wednesday at noon, 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 Helix 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 **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 *Helix Core 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 Helix server 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 316 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 Helix server database files only. Your versioned files might be stored in directories under the Helix server root, and must be backed up separately.

> **Tip**
> For more information about the location of versioned files, see the Tip in Step 4 at "Backup procedure" on the next page.

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 Helix server 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.

Helix server 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 Helix server 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 procedure

> **Tip**
> If your site is very large (gigabytes of `db.*` files), creating a checkpoint might take a substantial amount of time. Therefore, consider:
>
> - only creating a checkpoint at the end of each work week, while rotating the `journal` file during your nightly backup
> - performing checkpoints on a separate instance of your Helix Core p4d Server database. See the Support Knowledgebase article, Offline checkpoints.

To back up your Helix server installation, perform the following steps as part of your nightly backup procedure.

1. Make a checkpoint by invoking `p4d` with the `-jc` (journal-create) flag, or by using the p4 admin command. Use one of the following:

   On the host, where you might have a script that runs daily and also manages checkpoint files:

   ```
   $ p4d -jc
   ```

   or, on the client that is physically separate from the host:

   ```
   $ p4 admin checkpoint
   ```

   Because `p4d` locks the entire database when making the checkpoint, you do not have to stop the Perforce service during any part of the backup procedure.

2. Ensure that the checkpoint has been created successfully before backing up any files. If a disk crash occurs, it is important to know that the checkpoints you've been backing up are complete.

   Verifying either of the following:

   - `p4d -jc` (or `p4 admin checkpoint`) returns the value of `0`
   - the current journal file is truncated

   You can also use the command `p4d -jv` to verify the integrity of a checkpoint.

3. 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. The `.md5` file 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.

4. Once the checkpoint has been created successfully, back up:

   - the checkpoint file and its `.md5` file
   - the rotated journal file. If the checkpoint is *n*, the rotated journal is `journal.`*n-1*. See also "Journal files" on page 179.
   - the license file
   - the versioned files

Tip
OPTIONAL for backup:

- log files
- readonly clients - see "Using read-only and partitioned clients in automated builds" on page 251

There is no use case for backing up the following:

- `db.*` files
- server.locks directory

Note
There are rare cases 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:

- users obliterating files during backup, or
- users submitting files on Windows servers during the file backup portion of the process

Most sites are unaffected by these issues. Having Helix server 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.

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, Helix server is unable to access the files. Therefore, 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.

Tip
The version files to back up might be in one of the following:

- The directory with the same name as each depot containing submitted files in the P4ROOT - this is the default.
- The absolute path defined in the depot spec
- The location set by the server.depot.root configurable, if it is set, which is the case with a package installation that uses the `configure-helix-p4d.sh` shell script. This script:
  - can be used to set the the server.depot.root configurable to a location such as `/depots/archives` and to verify the location, use the `p4 configure show server.depot.root` command
  - sets journalPrefix to `../journals`, the location for checkpoints and rotated journals

> - sets up a nightly checkpoint by default. The crontab runs **`helix-p4d-maintenance.sh`** at **`4am`** system time. This script checkpoints any servers with **`MAINTENANCE=true`** in the **`p4dctl`** config file. It also removes any checkpoints and rotated journals older than 14 days. See "Helix Core Server Control (p4dctl)" on page 511.

5. If you have used the `p4 serverid` command to identify your server with a **`server.id`** file, the **`server.id`** file, which is in the server's root directory, must be backed up.

> **Tip**
> We recommend that administrators perform `p4 verify` weekly, rather than nightly. For large installations, the verification of files:
>
> - takes considerable time to run
> - puts the server under heavy load, which can impact the performance of other Helix server commands
>
> The command is:
>
> ```
> $ p4 verify //...
> ```
>
> or
>
> ```
> $ p4 verify -q //...
> ```
>
> The **`-q`** (quiet) option produces output only if errors are detected.
>
> By running **`p4 verify`**, you confirm whether the archive data on the server is correct. Regular use of **`p4 verify`** is good practice because it enables you to:
>
> - locate any corruption
> - determine whether or not the files restored from your backups following a crash are in good condition
>
> For more about the **`p4 verify`** command, see "Verifying files by signature" on page 27.

> **Tip**
> The instructions on this page are basic. If you organization has a large data set, see the reference implementation that features zero-downtime checkpoints, near-zero downtime upgrades, and more: the Server Deployment Package (SDP) at https://swarm.workshop.perforce.com/projects/perforce-software-sdp/.

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 Helix server 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 `p4d` server:

   ```
   $ p4 admin stop
   ```

   (You must be a Helix server 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
Version 2018.1

Starting with Version 2018.1, you no longer need to specify the `-z` option when restoring compressed journals and checkpoints. This is especially useful when restoring a compressed checkpoint and multiple journals in the same operation. For example:

`p4d -r . -jr checkpoint.42.gz journal.42 journal.43 journal`

Prior to version 2018.1

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 190 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 `p4d` server:

```
$ p4 admin stop
```

(You must be a Helix server 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

After you recover the database, you need to restore the versioned files according to your operating system's restoration procedures 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 the next page 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 Helix server command to examine how files in their client workspaces differ from those in the depot. If they run this command:
>
> ```
> $ p4 diff -se
> ```
>
> They are provided with a list of files in their workspace that differ from the files Helix server 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.

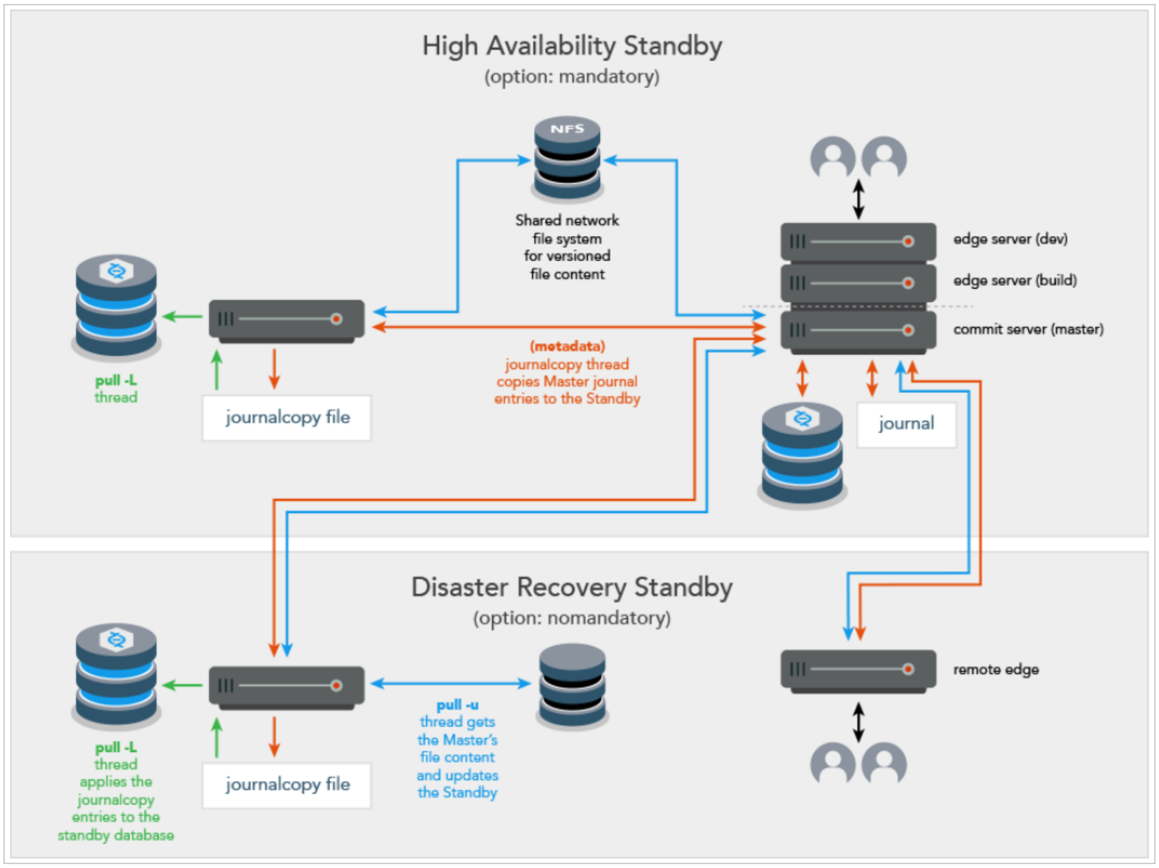
If any versioned files are reported as `MISSING` by the `p4 verify` command, you’ll know that there is information in the database concerning files that didn’t get restored. The usual cause is that you restored from a checkpoint and journal made after the backup of your versioned files (that is, that 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.

Failover

Failover is the process by which a standby server (or forwarding-standby server) replaces a server that provides standard, commit-server, or edge-server services. The server replaced during a failover is generally referred to as a "master" server.


High Availability Standby
(option: mandatory)
NFS
Shared network file system for versioned file content
edge server (dev)
edge server (build)
commit server (master)
(metadata) journalcopy thread copies Master journal entries to the Standby
pull -L thread
journalcopy file
journal
Disaster Recovery Standby
(option: nomandatory)
remote edge
pull -u thread gets the Master's file content and updates the Standby
pull -L thread applies the journalcopy entries to the standby database
journalcopy file

High Availability and Disaster Recovery

The Failover feature supports two scenarios:

- **High Availability (HA)**
  - The master can be configured as a master server, a commit server, or an edge server.
  - Typically, the standby server is in the same hardware rack as the master server
  - Typical use case: scheduled maintenance, but also possible if the master hardware fails

- Typically, the master server participates in the failover process:
  - disabling itself in an orderly fashion
  - waiting for the journalcopy of the remaining transactions to the standby
  - allowing the standby to stop the master

  > **Note**
  > If the master server does not participate in the failover, a check is made to ensure that the standby server to which failover is to occur has the `mandatory` option set. Without the participation of the master server, failing over to a `mandatory` standby server is required to ensure that the other replicas remain consistent with the new master server after failover. Consistency is assured because during production operations, metadata must be journalcopy'd by all `mandatory` standby servers before that metadata is replicated to the other replicas. Deploying one or more `mandatory` standby servers local to the master server is recommended. This is because journalcopy performance of the `mandatory` standby servers can affect the production replication to the other replicas.

- **Disaster Recovery (DR)**
  - Typical use case: due to a sudden catastrophe, the master server (and any HA standbys) are unable to operate.
  - Contact support for assistance with failing over to a non-mandatory standby server when the master server is inaccessible.

Consistency of the downstream replicas is assured for failing over when:

- **the master server participates**, in which case:
  - the standby server need not be a "mandatory" standby
  - the standby server's `journalcopy`, `pull -L`, and `pull -u` threads are an integral part of the failover
- **the master server does not participate** and the standby server is a "mandatory" standby, in which case only the standby server's `pull -L` thread is an integral part of the failover

Prerequisites for a successful failover

- The `p4 failover` command must be run on a server of Type `standby` or `forwarding-standby`. See the "Standby and forwarding-standby server" on page 406 topic.
- The standby (or forwarding-standby) server must be appropriately licensed for its new role following the failover. We therefore recommend that you submit a Duplicate Server Request.
- Make sure that monitoring (p4 monitor) is enabled for the new standby server (former master or commit server).

  - Monitoring must be enabled at server startup of the standby prior to running the **p4 failover** command. This is because the monitor subsystem is used to terminate the `journalcopy`, `pull -L`, and `pull -u` threads during the failover sequence.

- Open the server spec for each `standby` and `forwarding-standby` server. In the `ReplicatingFrom` field, enter the `serverID` of the server from which the standby server is journalcopy'ing.
- If an edge server is being failed over, the service user of the edge server should be logged into the commit (or master) server using the file specified by the P4TICKETS variable that is defined for the standby of the edge server. For example, issue the following command on the standby server that will become the new master:
  `p4 -E P4TICKETS=`*`directory`*`/.p4tickets -p `*`master`*`:`*`port`*` -u `*`service-user`*`:login`
- We recommend that a DNS alias point to the IP address of the master server. This allows the same DNS alias to point the new master server (former standby server).

> **Note**
>
> - To be prepared in case you might need to decide whether a failover operation is necessary, consider monitoring a target server by setting up "Triggering on heartbeat (server responsiveness)" on page 349.
> - If your scenario is a replica and a master with no standby, see the Knowledge Base article on "Failing over to a replica server".

Failing over to a standby or forwarding-standby

Failing over to a dedicated `standby` is generally faster than failing over to a forwarding-standby. For situations where failover completion is less time-critical, you might want to consider a `forwarding-standby`. See "standby" and "forwarding standby" in p4 server in *Helix Core P4 Command Reference*.

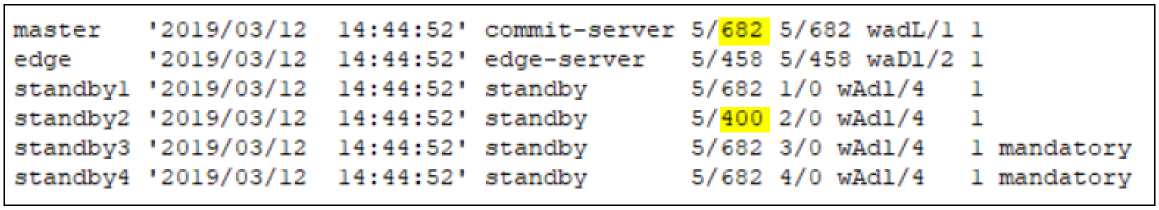
High availability with the mandatory server specification option

> **Important**
>
> A high availability standby within an existing installation should not be initially deployed as mandatory.

To deploy standby servers with minimal interruption to replication, make sure the `journalcopy` thread of the new standby server is caught up with the server from which is it journalcopying BEFORE you set the standby to `mandatory`. Follow this process:

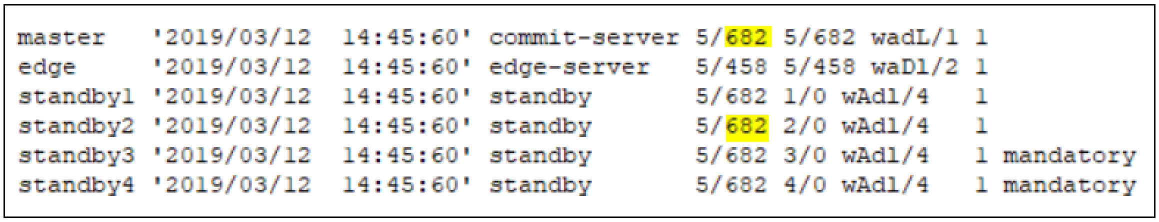
1. Deploy the standby with the default, which is `nomandatory`
2. To monitor the progress of the standby's journalcopy, on the server from which the standby is journalcopying, invoke p4 servers `-J`

In this example, we have invoked p4 servers `-J` on `master`, and we see that `standby2` has `400`, which does not yet match the `682` value on `master`:



```
master   '2019/03/12  14:44:52' commit-server 5/682 5/682 wadL/1 1
edge     '2019/03/12  14:44:52' edge-server   5/458 5/458 waDl/2 1
standby1 '2019/03/12  14:44:52' standby       5/682 1/0 wAdl/4   1
standby2 '2019/03/12  14:44:52' standby       5/400 2/0 wAdl/4   1
standby3 '2019/03/12  14:44:52' standby       5/682 3/0 wAdl/4   1 mandatory
standby4 '2019/03/12  14:44:52' standby       5/682 4/0 wAdl/4   1 mandatory
```

Later, again on `master`, that is, the server from which the standby is journalcopying, we invoke p4 servers `-J` again.
This example shows that that `standby2` has progressed to `682`, which matches `master` and indicates that `standby2` has a current journalcopy.



```
master   '2019/03/12  14:45:60' commit-server 5/682 5/682 wadL/1 1
edge     '2019/03/12  14:45:60' edge-server   5/458 5/458 waDl/2 1
standby1 '2019/03/12  14:45:60' standby       5/682 1/0 wAdl/4   1
standby2 '2019/03/12  14:45:60' standby       5/682 2/0 wAdl/4   1
standby3 '2019/03/12  14:45:60' standby       5/682 3/0 wAdl/4   1 mandatory
standby4 '2019/03/12  14:45:60' standby       5/682 4/0 wAdl/4   1 mandatory
```

3. Change the server spec for `standby2` to specify `mandatory`

   On the innermost master server, in the server specification for `standby2`, under Options, `mandatory` is now appropriate for a `standby` (or `forwarding-standby`) server. This option ensures that no replica has metadata that has not been copied to the journalcopy of all mandatory `standby` (or `forwarding-standby`) servers.

   If the master were unavailable, `standby1`, which is not a mandatory standby, could not be used for failover

   If the master is available, all four of the standbys could be used for failover.

Note
If the server from which failover is to occur is not participating in the failover (because the master is unavailable or the `-i` option causes the master to be ignored), the `p4 failover` command returns an error if it is running on a `standby` (or `forwarding-standby`) server that is not properly configured with the `mandatory` option.

Disaster recovery with the nomandatory server specification option

For disaster recovery failover, the remote standby typically has a server specification with the `Option:` field set to the default value, which is `nomandatory`. This is because the journalcopy performance of a `mandatory` standby can affect the speed of replication to the replicas of the master.

Potential data loss

If the master participates

- Any commands that were not completed when failover began might need to be executed again on the new master server.
- There should not be any data loss.

If the master does not participate

- Standby is `mandatory`
- Any commands that were not completed when failover began might need to be executed again on the new master server.
- The transactions that were done directly on the master prior to the failover that had not yet been journalcopy'd to the standby being used for the failover will be lost.
- To minimize data loss, the standby used for the failover should be the standby that was the most current with the master at the time of the failover. Typically, this is the standby that is in the same rack with the master.
  - The downstream replicas are consistent with the new master server
  - The downstream replicas will not have data loss relative to the new master server

Failover process

The Failover feature allows the super user to:

1. Get a report of whether conditions look good for a successful failover.

   > **Warning**
   > If the report indicates that the existing master server is still accessible and ignoring that server has been requested with the `-i` option, this could result in two separate servers, each of which is unaware of the other. This "split-brain" situation can produce inconsistencies that compromise the integrity of your data.

2. Initiate the failover process.
   a. This automatically stops the standby (or forwarding standby) server that will become the new master.
   b. During the failover process, the master server does not process any new commands and end-users get the "`failoverMessage`" (see the p4 failover command).
   c. A verification process ensures that recent file content was correctly replicated to the new master. See the p4 failover command for the `-v` option.

   d. During the failover process, the P4ROOT directory will get a new file named **`statefailover`**. This file is the last consistency point journalcopy'd by the standby immediately prior to the failover. This file will be deleted by the new master server when it is no longer needed.

      For example,
      `p4 failover`
      Make sure the preview looks OK.
      If so, then run
      `p4 failover -y`

3. Monitor the steps that are reported during the process. If the Failover process encounters an error, the process is designed to inform the superuser and to stop the failover process so that corrective action can be taken and a new attempt can occur.
4. If an error is encountered after the standby server has stopped the master server, the standby server will not restart the master server.
5. Verify, after the completion of a successful failover, that the former standby (or forwarding standby) has been restarted as the new master by issuing the p4 info command and checking the **`ServerID`** to ensure that it is the same **`ServerID`** that the previous master server used.
6. Following a successful failover, site-specific changes might be needed to use the new master server. It might be necessary to make DNS changes so that users and replicas can connect to the new master server. For example,
   - If you have a DNS alias set up, update the IP address of that DNS alias to point to IP address of new the master or commit server.
   - If you do not have a DNS alias set up,
     - change the P4TARGET environment variable on each replica or edge server by issuing the **`p4 configure show allserver`** command and issuing **`p4 configure set "`*`replica-name`*`#P4TARGET=`*`new-master-server`*`:`*`port-number`*`"`**
     - update your server specifications with the proper hostname and port number by issuing the **`p4 server`** ***`servername`*** command.

The end users can now issue new commands.

Note that you can "Failback after failover" on the facing page.

Configurables affected

The failover process:

- makes no changes to the configurables on the *original* master server
- can make changes to the following configurables for the *new* master so that the values are appropriate for the new environment:

client.readonly.dir	P4AUDIT
client.sendq.dir	P4JOURNAL
journalPrefix	P4LOG
pull.trigger.dir	P4TICKETS
server.depot.root	P4TRUST
server.locks.dir	P4ROOT
statefile	

Configurables and edge server

When failing over to a standby from an edge (or other replica) server, the updated configurables for the edge server will need to be manually changed on the commit server. This is because the update of the configurables cannot be propagated back to the commit (or upstream) server automatically, given that the edge server might, or might not, be participating in the failover.

Failback after failover

Failback reestablishes the relationship between the commit or master server and the standby server to what it was prior to failover. Failback is common when the failover was done for testing purposes. Failback is also a valid choice after failover for disaster recovery or maintenance.

After a successful failover, the original standby server has taken over as the new commit or master server.

To make a failback possible, ensure that the original commit or master server is:

- reconfigured as a standby before starting it
- assigned the `serverid` of a standby server, and that this `serverid` is different from the `serverid` of the current commit or master server

Reseeding the original commit or master server

Generally, it is a best practice to reseed the original commit or master from the new commit or master before performing the failback steps. Consider the following:

If the original commit or master did not participate in the failover ...	If the original commit or master participated in the failover ...
... the old commit or master's metadata might contain transactions that did not make it to the original standby at the time of failover. Those transactions could reappear at failback. To avoid this possibility, reseed the original commit or master from the new commit or master before performing the failback steps.	... reseeding might not be necessary. However, if you have any doubts about metadata integrity, the safest option is to reseed the original commit or master from the new commit or master before performing the failback steps.

Failback steps

At the new commit or master server

1. Verify that the standby is pulling from new commit or master server by issuing `p4 servers -J`
2. Check the result, which might be something like:

   ```
   commit '2019/07/09 16:41:36' commit-server 40/13642 40/13642 wadL/1 1
   ```

   ```
   standby '2019/07/09 16:41:31' standby 40/10000 40/10000 wAdl/4 1
   ```

   where `10000` is lower than `13642`, which indicates that the standby is not yet fully caught up with the commit or master server.
3. Wait a moment, then reissue `p4 servers -J` to verify that standby is fully caught up with the commit or master server. For example:

   ```
   commit '2019/07/09 16:41:36' commit-server 40/13642 40/13642 wadL/1 1
   standby '2019/07/09 16:41:36' standby 40/13642 40/13642 wAdl/4 1
   ```

At the new standby that was the original commit or master server

1. Issue the failover command: `p4 failover`
2. Follow the steps at "Failover" on page 191.

Monitoring the server

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 Helix server.

By default, Helix server rejects commands when free space on the filesystems housing the `P4ROOT`, `P4JOURNAL`, `P4LOG`, or `TEMP` falls below 250 megabytes. To change this behavior, set the `filesys.P4ROOT.min` (and corresponding) configurables to your desired limits:

- `filesys.P4ROOT.min`
- `filesys.P4JOURNAL.min`
- `filesys.P4LOG.min`
- `filesys.TEMP.min`
- `filesys.depot.min`

If the user account that runs the Helix 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.

For more information, see Configurables in the *Helix Core P4 Command Reference*.

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:

Format	Meaning
nnn	A plain number, used as is.
*nnn*K	A number in kilobytes `$ p4 configure set filesys.P4TEMP.min=100K`
*nnn*M	A number in megabytes `$ p4 configure set filesys.P4ROOT.min=10M`
*nnn*G	A number in gigabytes. `$ p4 configure set filesys.P4JOURNAL.min=1G`
*nnn*T	A number in terabytes.
nnn%	A number as a percentage of the current space. To reserve ten percent of the total disk space 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` reflects the disk space required when files are synced to a client workspace, but can provide a useful estimate of server-side disk space consumption.

Monitoring processes

Use the **`p4 monitor`** command to observe and control Helix server-related processes running on your Helix server machine.

Enabling process monitoring

Server process monitoring requires minimal system resources, but you must enable process monitoring for `p4 monitor` to work.

For example, to monitor active commands, set the `monitor` configurable to `1`:

```
$ p4 configure set monitor=1
```

To include idle processes, set to a value higher than `1`.

Valid values for the monitor configurable are:

- `0`: Server process monitoring off. (Default)
- `1`: monitor active commands
- `2`: active commands and idle connections
- `3`: sames as `2`, but also includes connections that failed to initialize (stuck at the Init() phase)
- `5`: sames as `2`, but also includes a list of the files locked by the command for more than one second
- `10`: same as `5`, but also includes lock wait times
- `25`: sames as `10`, except that the list of files locked by the command includes files locked for *any* duration

Note
The command **`p4 monitor -ael`** includes

- the command arguments (**`-a`**)
- the environment (**`-e`**)
- long-form output (**`-l`**), including the username and argument list.

If your rejected client version still appears in the output, ensure the **`rejectList`** setting is correct. See "Rejecting client connection requests" on page 81.

Note

- Regarding `5`, `10`, or `25`, for Linux and MacOS systems, see the p4 monitor topic on the `-L` option.
- Microsoft Windows does not have the `lsof` utility to list open files, so `5`, `10`, or `25` are not relevant to Windows.

Important

Setting monitor to a valid non-zero value activates db.monitor.interval. For example,

1. Set a valid non-zero value for monitor, such as `p4 configure set monitor=1`
2. (Optional): If you want a different monitoring interval than the default 30 seconds, set the db.monitor.interval configurable with a command such as `p4 configure set db.monitor.interval=120`

Tip

See "Terminating blocked processes" on page 242.

Enabling idle processes monitoring

By default, `IDLE` processes, which are often associated with custom applications based on the C/C++ 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 Helix 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 Helix server 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 Helix 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 Helix server application (and version string or API protocol level), *`IP-address`* is the IP address of the user's Helix server 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 Helix server administrators and superusers can use the `-a`, `-l`, and `-e` options.

Diagnostic flags for monitoring the server

Using diagnostic flags can help you monitor the server.

Any user commands that exceed certain thresholds for resource usage (such as CPU, lapse time, database I/O, network I/O) automatically get logged into the server error log specified by P4LOG. Trace output appears in the log file, and shows the date, time, username, IP address, and the command for each request the server processes.

Performance Tracking

Performance tracking is on by default (determined by the number of users shown in the server license file) but can be turned off or adjusted with the `p4 configure set track=x` command.

Performance Tracking

Level	Description
`0`	Turn off tracking
`1`	Track all commands
`2`	Track excess usage for a server with less than 10 users
`3`	Track excess usage for a server with less than 100 users
`4`	Track excess usage for a server with less than 1000 users
`5`	Track excess usage for a server with more than 1000 users

Command Tracing

Command Tracing is on by default but can be turned off or adjusted with the `p4 configure set server=x` command:

Command Tracing Levels

Level	Description
`0`	Turn off tracking
`1`	Include the start information for each command
`2`	Include the start and stop information for each command
`3`	Add a "compute end" message for certain commands
`4`	Include errors sent to the client to the server log.

Setting the diagnostic flags

To modify the behavior of command tracing or performance tracking, use the p4 configure command. For example:

```
$ p4 configure set server=3
```

> **Tip**
> Before you activate logging, make sure that you have at least the minimum required disk space (see filesys.P4LOG.min in Helix Core P4 Command Reference) and be aware that you might need more.

Setting server debug levels on a Helix server server (`p4d`) has no effect on the debug level of a Helix Proxy (`p4p`) process, and vice versa.

> **Note**
> The highest levels of the Helix server command tracing and tracking flags are typically recommended only for system administrators working with Perforce Technical Support to diagnose or investigate problems.

To enable both `server` and `track` flags:

Issue these two commands:

```
p4 configure set track=1
p4 configure set server=3
```

> **Note**
> - Diagnostic flags can also be set using P4DEBUG or on the server command line using the `-v` option.
> - For additional information, see the Knowledge Base article, Interpreting server log files.

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.

Logging

Logging commands

You can use the following commands to work with logs and structured 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 logschema`	Return a description of the specified log record type.
`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`**).

For information about logging to a single file, see P4LOG in *Helix Core P4 Command Reference*.

For information about structured logs, see "Structured logs" on page 208.

Examples of possible log entries

The following is a subset of possible log entries:

Entry	Meaning
`user-transmit`	`user-transmit -t <taskID> [-b batch -s batchsize -r]` Processes spawned by parallel sync, submit, or shelve that transfer batches of files in parallel. The arguments correspond to the `batch` and `batchsize` arguments of the parallel sync, submit, or shelve command. ■ The internally-generated `-r` argument indicates that the parallel submit or shelve transfers from the client to the server, rather than server to client (like sync) ■ The `-t` argument is internal
`rmt-Journal`	Used by a **p4 pull** thread on a replica server to retrieve journal records that contain metadata from a master. ■ in the replica server log, you might see the pull processes ■ in the master log, you might see `rmt-Journal` entries corresponding to the metadata being pulled from the master to the replica server
`rmt-FileFetch`	Used by `p4 pull -u` on an replica server to retrieve archive files, or by parallel submit from an replica server to transfer archive files from the replica server to the master. ■ In the master log, you might see `rmt-FileFetch` entries from the `pull -u` commands running on replica servers. ■ In the Edge Server log, you might see `rmt-FileFetch` entries during parallel submit from the pull command running on the Commit Server to get the archives onto the Commit Server

Auditing user file access

Helix 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 the server is started with the `-A` ***`auditlog`*** flag (see "General options" on page 501 in "Helix Core server (p4d) Reference" on page 499).

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
2020/05/09 09:52:45 karl@nail 192.168.0.12 diff //depot/src/x.c#1
2020/05/09 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 Helix 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.

Structured logs

Helix server can be configured to write log files in a structured (**`.csv`**) format.

Structured log files:

- contain more detail than conventional log files
- make it easier to import the data into other tools, such as spreadsheets, for further reporting and detailed analysis

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.

Enable and configure structured logging

To enable structured logging, set the serverlog.file.*N* configurable to the name of the file.

> **Note**
> Enabling all structured logging files can consume considerable amounts of disk space. To manage the size of the log file and the number of log rotations, see "Structured logfile rotation" on page 210.
>
> The value you specify for ***`N`*** cannot exceed `500`.

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, triggers, and more)
`audit.csv`	Audit events (audit, purge)

Filename	Description
`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.
`ldapsync.csv`	p4 ldapsync events, such as when: ■ a user is added, updated, or removed ■ a user is added or removed from a group
`route.csv`	Log the full network route of authenticated client connections. Errors related to `net.mimcheck` are also logged against the related hop.
`track.csv`	Command tracking (track-usage, track-rpc, track-db)
`triggers.csv`	Trigger events.
`user.csv`	User events; one record every time a user runs `p4 logappend`.

Structured Log Configuration

Use **`p4 configure set`** to configure the log file name and optionally auto-rotation and retention policies. For example, to set up an **`errors.csv`** structured log to auto-rotate when it reaches 100MB and to retain the last 10 logs:

```
p4 configure set servername#serverlog.file.1=errors.csv
p4 configure set servername#serverlog.maxmb.1=100
p4 configure set servername#serverlog.retain.1=10
```

Auto-rotation and retention policies are optional but recommended to manage the size of structured logs. If you do not specify a **`servername`** with your `p4 configure` **`set`** command, the configurable is set for all servers in a Helix distributed environment. This includes edge and replica servers unless they override the global configuration with their own configurable setting.

Structured log files are automatically rotated on checkpoint, journal rotation, overflow of associated serverlog.maxmb.*N* limit if configured, and the p4 logrotate command. To disable structured log rotation after checkpoint or journal rotation, set the dm.rotatelogwithinjnl configurable to **`0`**.

The **`p4 -ztag`** `logstat` **`-s`** command is used to provide a summary of the configured structured logs. The configuration of structured logs is dynamic. After p4 configure set is run, no server restart is required and the server generates the log and starts using it when it processes the next server command. The name of a structured log automatically configures the events that log will capture.

Note
Files do not have to be set in consecutive order:

```
$ p4 configure set serverlog.file.1=audit.csv
$ p4 configure set serverlog.file.2=auth.csv
$ p4 configure set serverlog.file.4=track.csv
$ p4 configure set serverlog.file.3=triggers.csv
```

Structured logfile rotation

Rotating structure logfiles is a best practice to allow the analysis of recent events while also limiting the total volume of log data.

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.

2020.1 Structured logging improvements

"Protocol levels of server and client by server release number" on page 32 shows the protocol levels for every release, and the 2020.1 release has the following improvements for structured logging:

- All event types have a version `50`, which adds a unique command identifier and the current `serverId` after the command number field (5th column).
- For commands that result in log events being written on multiple servers, the unique command identifier propagates across the server to allow those events to be matched. Examples include submits from edge servers, replica forwarded commands, remote depot access, and P4AUTH.
- The `11.50` trigger event type has two additional fields:
  - a trigger type of `trigger`, `extension`, or `bgtask` for background task
  - an execution lapse time
- Trigger arguments are now separated by the colon `:` character to match the command arguments format.
- The serverlog.version.N configurable can be used to pin a structured log file to the format of a specific server version. For example:
  - to retain the 2019.2 structured log events format, set this configurable to `49`
  - to retain the format of 2019.1, set this configurable to `48`

Detailed logging of server activity

Using P4LOG

The P4LOG environment variable allows you to specify the name and path of the file to which Helix server errors are written.

The default level of verbose logging is governed by hard-coded thresholds. Verbosity of the log file produced by Perforce Server can be set by invoking p4d with the -vtrack flag.

The p4d server produces diagnostic output to help identify performance problems and is on by default but can be turned off or adjusted with the `-vtrack=x` flag to the server. Any user commands that exceed certain thresholds for resource usage (CPU, lapse time, database I/O, network I/O, among other things) automatically get logged into the server error log P4LOG. The levels that can be set with `-vtrack=x` are:

- `0` turn off tracking
- `1` track all commands
- `2` track excess usage for a server < 10 users
- `3` track excess usage for a server < 100 users
- `4` track excess usage for a server < 1000 users
- `5` track excess usage for a server > 1000 users

If `-vtrack` is not provided on the server command line or set with P4DEBUG, the tracking level is computed from the number of users listed in the server license file.

The exact format of the tracking output is not documented, and subject to change.

Unless a Perforce Server is started using the `-v track=1` option, it is not guaranteed to log every command. To ensure that every command is logged (not just those commands exceeding the default performance thresholds), set the `-v server=1` flag in the p4d invocation for the installation. Note that multiple `-v` flags can be added to a p4d start command. For more information on server trace flags, see "Diagnostic flags for monitoring the server" on page 204.

Deciphering the first lines

Perforce server info:

```
2018/08/02 14:25:10 pid 16256 cpflaum@work2 127.0.0.1
[p4/2018.2/LINUX26X86_64/165458] 'user-sync -p //depot/WORK/...'
--- lapse 109s
```

The first two lines provide the information about what command was run, when it was run, what server was run on, and by whom. The lapse information on the third line is wall clock time, and shows that the command took 109 seconds to run.

The usage information format is based on the output of the `getrusage` command, common to many Linux systems.

Deciphering usage

```
--- usage 9383+3937us 0+0io 0+0net 0k 29916pf
```

9383+3937us

Time spent executing user instructions + Time spent in operating system code on behalf of processes. Times are represented in milliseconds (us = user and system)

0+0io

The number of times the file system had a 512-byte physical read from the disk on behalf of processes + the number of times the file system had a 512-byte physical write on behalf of processes.

The block reads, that is the "X" of the "X+Yio, are the physical reads charged to the process. A value of zero means that all data that was needed by the command was in the filesystem cache. Note that the block reads that are reported are not specific to any one table.

If the io is seen in a parallel sync such as

```
'user-transmit -t 61063 -b 8 -s 524288'=20
--- usage 56739+97512us 3448+21128824io 0+0net 12064k 0pf
```

this would be interpreted as a large number of physical disk writes (21,128,824 512-byte disk blocks) which includes pages written to db.have to update the client's have list after receiving files and pages written to db.sendq removing records corresponding to the files that this transmit thread sent to the client. The physical disk writes are those that occur prior to the fsync() call on a modified db.*.

0+0net

IPC (Inter Process Communication) messages received on behalf of processes + IPC message sent on behalf of processes.

0k

The maximum number of kilobytes of physical memory that processes used simultaneously.

29916pf

The number of page faults that were serviced by doing I/O.

Deciphering RPC (remote procedure call)

```
--- rpc msgs/size in+out 3+431751/0mb+1417mb {himarks 64835/64835}
```

The above output example displays the number of RPC messages and size of the data transfers to and from the disk, in megabytes. In this case, a 1.417 GB was transferred from the server to the local client during a sync command. While 3 messages were sent back, the size of those messages was small enough to be a rounding error, thus zero megabytes. A submit command would show the opposite condition, with a large number of messages and data being sent to the server, with fewer coming back to the client.

Optionally, "himarks" is displayed to show the send/receive window size used for the RPC messages. This can be useful to catch some client issues where the window sizes (in bytes) may be set incorrectly by the OS. For example, an issue with older Perforce Windows clients with an artificially low send window size would appear as:

```
msgs/size in+out 0+2/0mb+0mb himarks 64835/2000
--- rpc receive errors, duplexing F/R 0/0
```

Deciphering lock times

This section concerns how many files and revisions were scanned in affected database files, and information about file locking on the Perforce server.

Read Locks

A read lock still allows other processes to read the particular table in the Perforce database. However, any write lock must wait for the read lock to complete.

Below is a command to find all read locks held for more than 10 seconds. Replace the `<log>` with the log filename. The results will be stored in file: `longReadHeldTimes.log`. You can then search the log for these times to find the processes that were holding read locks.

```
grep "total lock wait+held read/write " <log> | awk '{print $6}' |
awk -F/ '{print $1}' | awk -F+ '{print $2}' | sed s/ms//g | sort -
unr | sed '/.\{5\}/!d' > longReadHeldTimes.log
```

Write Locks

A write lock blocks any other processes from reading or writing to the particular table in the Perforce database.

If a process is waiting its turn for an action to complete, it is a victim. If a process locks the database table and other processes must wait, then it is a culprit. If a large number for read or write lock wait time is seen, look for an earlier corresponding large write lock hold time. This will identify the process holding other processes off.

Below is a command to find all write locks held for more than 10 seconds. You will need to replace the <log> with actual log filename. The results will be stored in file: longWriteHeldTimes.log. You can then search the log for these times to find the processes that were holding write locks.

```
grep "total lock wait+held read/write " <log> | awk '{print $6}' |
awk -F+ '{print $3}' | sed s/ms//g | sort -unr | sed '/.\{5\}/!d' >
longWriteHeldTimes.log
```

Good performance Example

```
--- db.resolve
---   total lock wait+held read/write 0ms+252ms/0ms+0ms
```

No time was spent waiting for the db.resolve table locks, and the process held a read lock for 252ms. The lock is only held for a fraction of of a second and therefore is not tying up the Perforce db.resolve database file.

Slow performance example

```
--- total lock wait+held read/write 23763ms+463ms/582568ms+343ms
```

The process was blocked for 23.763s waiting for a read lock (so it is a victim of another process holding the table).

The process held a read lock for 0.463 seconds (a relatively fast operation).

The process waited for 582.568 seconds before it could issue its own write lock (so now it is very much a victim of one or more processes that could be holding locks on multiple tables).

The process wrote for 0.343 seconds (still a fairly fast operation).

Therefore this process is not causing a performance slowdown, but it was a victim. As an example, the user might believe that Perforce was stalled or crashed during this time, however the server is still running, albeit very slowly. This is typical of a server undergoing a heavy usage spike, and will often "unwind" when the load drops.

Total Lock versus Max Lock

The total lock time is the total time that all locks of each type (read or write) for the table were blocked (wait) and held for the operation. The max lock time is the longest time that any one lock of each type for the table was blocked and held for the operation so it can be seen if one lock used up most of the time. If no more than one lock of each type was taken for the table, no "max lock" tracking entry is reported.

For example, the **`db.monitor`** output below lists both **`total`** and **`max`** locks:

```
...
$ p4 -Ztrack changes -m366 //depot/blaster/engine/... | grep "\-\-\-"
...
--- db.monitor
---   pages in+out+cached 3+4+2
---   locks read/write 15/2 rows get+pos+scan put+del 15+0+0 1+1
---   total lock wait+held read/write 0ms+1ms/0ms+0ms
---   max lock wait+held read/write 0ms+1ms/0ms+0ms
```

For this changes command, 15 read locks and 2 write locks were taken for

db.monitor. For these locks:

The time that the 15 read locks were blocked totaled 0ms.

The time that the 15 read locks were held totaled 1ms.

The time that the 2 write locks were blocked totaled 0ms.

The time that the 2 write locks were held totaled ~0ms.

The maximum time that any one of the 15 read locks was blocked was 0ms.

The maximum time that any one of the 15 read locks was held was 1ms.

The maximum time that any one of the 2 write locks was blocked was 0ms.

The maximum time that any one of the 2 write locks was held was ~0ms.

Deciphering change/changelistnumber()

You may see an entry similar to change/3082654(W). It is an exclusive server lock on change 3082654 to ensure that there is no concurrent access to shelf 3082654 while it is being worked on.

```
Perforce server info:
2018/10/19 09:04:39 pid 11571 bruno@gabriel
```

```
172.91.82.126/172.91.10.167
p4/2018.1/LINUX26X86_64/1559260 (brokered)] 'user-shelve -c 3082654 -
d'
--- lapse 4427s
--- change/3082654(W)
---   total lock wait+held read/write 0ms+0ms/0ms+4427001ms
```

Buffering

In the course of processing a command the server reads and writes data pages to and from a buffer cache:

```
--- db.revhx
---   pages in+out+cached 182412+0+64
```

`in` = number of pages read into the Perforce buffer/cache.

`out` = number of pages written out of the Perforce buffer/cache.

`cached` = number of pages retained in the Perforce buffer/cache.

The "cached" value is not an indication that the db.* table is in the filesystem cache. Rather, the "cached" value reflects the maximum number of pages in the (small) process-private memory for each db.* file that is used to minimize reads by caching pages that might be read more than once by only that process.

Table scanning

"Get" fetches a single row given a key. "Position" and "Scan" work together to fetch many rows -- the key is given to "Position", and then "Scan" returns rows after that key.

For example, "Get" is used to get the db.domain record for a client. "Position/Scan" are used to get all db.have records for a client.

```
--- db.revhx
---   locks read/write 1/0 rows get+pos+scan put+del 0+1+26635 0+0
```

The database only had to position once and scanned 26,635 revisions. If you see a lot of positioning to read relatively few rows (records), you might need to examine your protection table, client mappings, and command structure. Typically the fewer positions per file, the more efficient the process.

The -Ztrack flag

You can also use the -Ztrack flag on the p4 command-line to give database usage and locking information:

`p4 -Ztrack submit`

Once the command is completed the resulting database statistics are displayed in the same format as the log file.

Additional tools

To help analyze your server and log files, consider using:

- TRACK2SQL - a publicly available script that will create a mysql database of process information from a log file. This can be useful for identifying large processes that may affect overall server performance.
- Simple P4D Log Analysis - a simple command to analyze vtrack results.

If you have questions about your server performance and want assistance in your log file analysis, contact Perforce support.

Using structured logging

Structured server logs record information about server and user activity. Logs are written in comma separated value (CSV) format which can be processed by external tools or natively using the p4 logparse command. Structured logs can be configured to easily manage automatic log file rotation and log retention policies.

Structured server logs are intended to supplement standard Helix Server logs, not replace them.

Structured server logs:

- Record information about server and user activity
- Are written in comma separated value (CSV) format to facilitate parsing with the p4 logparse command
- Can be verbose and grow rapidly, so we recommend that you establish a process for automatic log file rotation and a policy for log retention

Versioned Log Schema

With Helix Server **2019.2** and later, structured logs have a versioned schema that allows new versions of existing events to be added. Updated versions of events are represented by having the server protocol level included after a period in the event type field. For example, `2.49` is associated with the 2019.2 release. See "Protocol levels of server and client by server release number" on page 32).

The serverlog.version.*N* configurable can be used to pin a structured log file to a specific server version's format. For example, to retain the 2019.2 structured log events format, set the serverlog.version.N configurable to `49`.

Event types with new versions in Helix Server 2020.1

All event types have a new version `N.50` in the **2020.1** release which adds a unique command identifier and the current serverId after the command number field (5th column). For commands that result in log events being written on multiple servers, the unique command identifier will have been propagated to allow those events to be matched up. Examples include submits from edge servers, replica forwarded commands, remote depot access, and P4AUTH.

The `11.50` trigger event type has two additional fields, `f_triggerType` which is one of `trigger`, `extension`, or `bgtask`, and `f_triggerLapse`, which records the trigger execution lapse time. Trigger arguments are now separated by colon (:) characters to match the command arguments format.

Event types with new versions in Helix Server 2019.2

2.49 - CommandEnd

6.49 - Audit

8.49 - NetworkPerformance

9.49 - DatabasePerformance

To retain the 2019.1 structured log events format, set the `serverlog.version.N` configurable to 48.

Structured Server Log Records

All structured server log records contain an event type written as the first field in the log record.

Prior to Helix Server 2019.2, the event type was an integer value.

With 2019.2 and later, the event type includes the server protocol level after a period in the event type field, such as `2.49` for a command-end log event written by a 2019.2 Helix Server.

Events are applicable to server versions that support structured logging (**2012.1 and later**), except where ** indicates that **2019.2 or later** is required.

Structured Log Events

Type	Name	Description
0	command-start	Command start
1	command-compute	Compute end
2	command-end	Command end; Lapse time and termination reason (error, auth, max*, monitor, etc) * *
3	errors-all	All errors

Type	Name	Description
4	errors-failed	Failed errors - User errors ('user' has not been enabled, Invalid option:, Submit failed)
5	errors-fatal	Fatal errors - System errors (Database open errors, Operation 'open' failed)
6	audit	Audit events (p4 sync, p4 archive, ...)
7	track-usage	Performance usage tracking
8	track-rpc	Network performance tracking; Includes send/receive errors and duplex stats * *
9	Network performance tracking; Includes send/receive errors and duplex stats * *	Database performance tracking; Includes lock times, including peek * *
10	user	User events from p4 logappend
11	trigger	Trigger events
12	event	Server startup, shutdown, restart, checkpoint, journal rotate
13	purge	purge revision (p4 obliterate)
14	network-estimates	Network estimates
15	integrity	Replica integrity checking events
16	auth	Login events
17	route	Network route of client connections
18	audit-size	Audit events including file revision's size
19	ldapsync	p4 ldapsync events
20	action	Primarily stores insert, update and delete actions such as: depot, change, submit
21	RemoteDatabasePerformance	logs tracking data for remote database access * *
22	ServerLockPerformance	logs the failed lock attempts prior to a blocking lock being taken * *

Type	Name	Description
23	DatabaseBlockingLocksTaken	logs the failed lock attempts prior to a blocking lock being * *

Structured Log Record Fields are based on event type

Events are applicable to server versions that support structured logging (**2012.1 and later**), except where ** indicated that **2019.2 or later** is required.

Field	Description	Event Types
action	Action	ldapsync purge
action	Audit action	audit
address	Location of remote database	RemoteDatabasePerformance **
args	Command arguments	command-compute command-end command-start RemoteDatabasePerformance * *
attempts	Number of lock attempts prior to blocking	DatabaseBlockingLocksTaken * *
client	Client workspace	All
cmdident	Command identity * *	All
cmdno	Command number	All
context	Context for trigger	trigger
date	Date	All
dbName	Database table name	track-db
deletes	Database deletes	track-db
eventCode	Event code	event
eventInfo	Event info	event
eventtype	Event type	All
file	File	purge
file	File name	audit
filesize	File revision size	audit-size

Field	Description	Event Types
func	Command	All
gets	Database gets	track-db
group	Group name	ldapsync
host	Client host	All
io_in	I/O reads	track-usage
io_out	I/O writes	track-usage
lapse	Lapse Time	command-end * *
lockName	Lock name	ServerLockPerformance * *
lockTarget	Lock locations	ServerLockPerformance * *
lockType	Lock type: - Read - Write	ServerLockPerformance * *
maxrss	Max physical memory	track-usage
net_in	IPC in	track-usage
net_out	IPC out	track-usage
page_faults	Page faults	track-usage
pagesCached	Pages cached	track-db
pagesIn	Pages in	track-db
pagesOut	Pages out	track-db
pid	Process ID	All
pipeProbe	Number of pipe probes sent remote location	RemoteDatabasePerformance * *
pipeRows	Number of DB rows sent to remote location	RemoteDatabasePerformance * *
pipeTime	Time taken to transmit data to remote location	RemoteDatabasePerformance * *
positions	Database positions	track-db
prog	Program	All

Field	Description	Event Types
readLocks	Read locks	track-db
reason	Reason for event termination: ■ standard error ■ terminated by monitor command ■ authentication failed ■ maxLimit reached	auth * * command-end * * ldapsync* *
recvBytes	Receive bytes	track-rpc
recvCount	Receive count	track-rpc
recvTime	Receive time	track-rpc
reorderIntl	Reorder internal	track-db
reorderLeaf	Reorder leaf	track-db
result	Result	ldapsync
rev	File revision	audit
rev	Revision	purge
rpc_hi_mark_fws	Highmark forward	track-rpc
rpc_hi_mark_rev	Highmark reverse	track-rpc
scans	Database scans	track-db
sendBytes	Send bytes	track-rpc
sendCount	Send count	track-rpc
sendTime	Send time	track-rpc
serverid	Server ID	All **
severity	Severity level	errors-all errors-failed errors-fatal
stime	System time	track-usage

Field	Description	Event Types
subcode	Subsystem code	errors-all errors-failed errors-fatal
subsys	Subsystem	errors-all errors-failed errors-fatal
targetID	Target ID	trigger
targetType	Target type	trigger
text	Error text	errors-all errors-failed errors-fatal
timer	Timer	track-usage
timestamp	Unix timestamp	All
timestamp2	High-precision timestamp	All
totalLockHeldRead	Total hold time on read lock	All
totalLockHeldWrite	Total hold time on write lock	ServerLockPerformance **
totalLockWaitRead	Total wait time on read lock	ServerLockPerformance **
totalLockWaitWrite	Total wait time on write lock	ServerLockPerformance **
trackType	Server tracking type	RemoteDatabasePerformance ** ServerLockPerformance **
trackType	Tracking type	track-db track-rpc track-usage
triggerAction	Trigger action	trigger
triggerData	Trigger data	trigger
triggerLapse	Trigger lapse time	trigger **
triggerType	Trigger type	trigger **
user	User	All
user	User Name	ldapsync

Field	Description	Event Types
utime	User time	track-usage
version	Program version	All
writeLocks	Write locks	track-db

Log record details

For a specific log record, use the event type and p4 logschema to get details on the specific fields in the record. For example, an event record (type 12) from the log:

```
12,1474054242,775358804,2016/09/16 12:30:42 75358804,21204,4,shutdown
p4 -F '%f_name%' logschema 12
f_eventtype
f_timestamp
f_timestamp2
f_date
f_pid
f_eventCode
f_eventInfo
```

Integrity events

The integrity event type is recorded in the `f_integrityEvent` field.

0	Unknown
1	VerifyResults
2	Unload
3	UnloadResults
4	ScanStart
5	ScanEnd
6	ScanResults
7	ChangeVerify
8	ChangeResults
9	TableResults

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 Helix server administrators and superusers to force certain operations unavailable to ordinary users. Helix server administrators can use this flag with `p4 branch`, `p4 change`, `p4 client`, `p4 job`, `p4 label`, and p4 unlock. Helix server superusers can also use it to override the `p4 user` command.

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
`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

Command	Syntax	Function
`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 the next page 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 230 for additional information about this option.

- **Using remote depots**

  This option enables independent organizations with separate Helix server 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 "Working with depots" on page 109.

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.

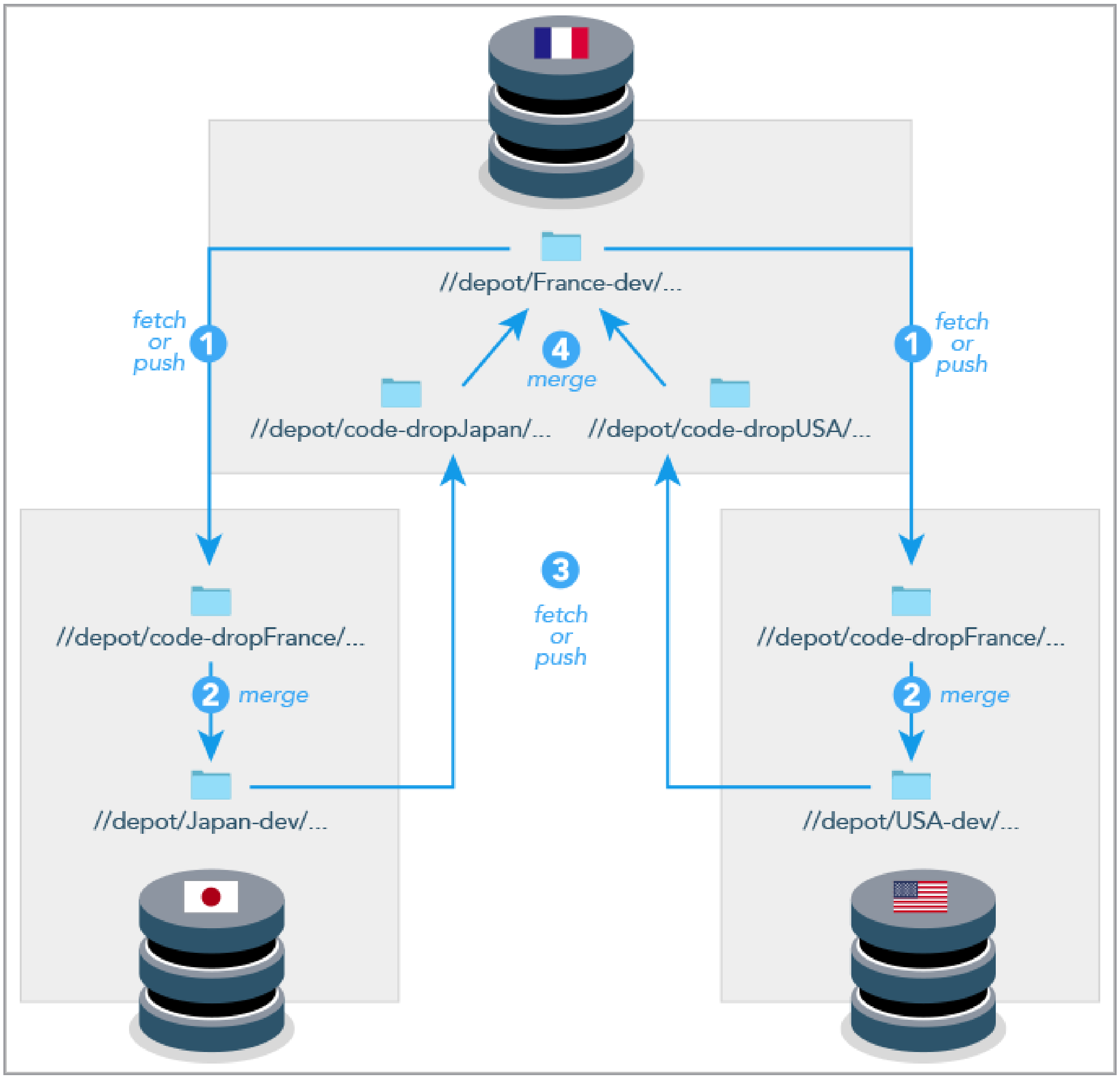
The gaming company Ukko Productions has offices in France, Japan and the United States. Each site is responsible for a different part of the gaming code; each does development on the section of code or "component" for which it is responsible. This work happens on the office’s Helix server, in a depot directory called `dev`. `dev` will contain locally submitted changes.

Let’s suppose France is working on a widget which is used by the developers in Japan and the United States. First, France makes the widget code available to Japan and the United States by dropping the code — using the `p4 push` into drop directories on the servers in Japan and the United States (see "1" in the figure below). (Alternatively, the Japan and United States developers could use the `p4 fetch` to copy France’s code into their drop directories.) The Japan and United States development teams can then merge the France 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 France developers.

If developers in the US and Japan have a subset of changes they do want to share with France, they use `p4 push` to copy this code into a special drop location on the France server — one location for Japan and one for the United States. (See "3" in the figure below). (Alternatively, France could use the `p4 fetch` to obtain the code and drop it into the appropriate locations.) The France developers can then merge the Japan 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 — Japan's, the United States' and France'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 section "Understanding Remotes" in *Using Helix Core Server for Distributed Versioning*.)

Because the Japan developers are fetching from or pushing to France's server, their server's remote spec would look as follows:

```
RemoteID: ServerFrance
Address: ServerFrance:1666
DepotMap:
  //depot/code-dropA/... //depot/France-dev/...
  //depot/Japan-dev/... //depot/code-dropS/...
```

Because the United States developers are fetching from or pushing to France's server, their server's remote spec would look as follows:

```
RemoteID: ServerFrance
Address: ServerFrance:1666
DepotMap:
  //depot/code-dropUSA/... //depot/France-dev/...
  //depot/USA-dev/... //depot/code-dropS/...
```

Because the France developers are fetching from or pushing to Japan, their server's remote spec would look as follows:

```
RemoteID: ServerJapan
Address: ServerJapan:1666
DepotMap:
  //depot/code-dropS/... //depot/Japan-dev/...
  //depot/France-dev/... //depot/code-dropA/...
```

Because the France 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/France-dev/... //depot/code-dropUSA/...
```

Code drops without connectivity

Helix 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 Helix server.

Managing users

This section describes the three types of Helix server 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 "Securing the server" on page 122.

User types

There are three types of Helix server users: `standard` users, `operator` users, and `service` users.

- A `standard` user is a traditional user of Helix server.

  Standard users are the default, and each standard user consumes one Helix server license.
- An `operator` user is intended for human or automated system administrators.

  An `operator` user does not require a Helix server license.
- A `service` user is used for server-to-server authentication in the context of remote depots and multi-server environments. See "Remote depots and multi-server development" on page 115.

  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, Helix server creates a new user record in its database whenever a command is issued by a user who does not exist. Helix server 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 Helix server licenses.

A service user can run 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

Note
Although a service user cannot run p4 pull directly on the command line, the service user on a replica automatically runs this command to retrieve metadata and archive content (versioned files) from the master.

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`:

```
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 118.

Operator users

Organizations whose system administrators do not use Helix server 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 Helix Core server, rather than the development of software or other assets on the server.

An `operator` user does not require a Helix server license, and can run only the following commands:

`p4 admin checkpoint`	`p4 admin journal`	`p4 admin restart`
`p4 admin stop`	`p4 configure`	`p4 counter (including -f)`
`p4 counters`	`p4 dbstat`	`p4 dbverify`
`p4 depots`	`p4 diskspace`	`p4 info`
`p4 jobs (including -R)`	`p4 journaldbchecksums`	p4 lockstat
p4 login	p4 logout	p4 logappend
p4 logparse	p4 logrotate	p4 logschema
p4 logstat	p4 logtail	`p4 monitor`
`p4 passwd`	`p4 ping`	`p4 pull` (including `-lj`)

p4 serverid	p4 servers	p4 user
p4 verify		

Preventing automatic creation of users

> **Warning**
> By default, Helix server creates a new user whenever a previously unknown user invokes any command that can update the repository or its metadata. When executed by a nonexistent user, most Perforce commands cause a user to be created. You can control this behavior by setting the `dm.user.noautocreate` configurable with the **`p4 configure`** command. For greatest security, we recommend that only the Helix server superuser be allowed to create new users:
>
> ```
> $ p4 configure set dm.user.noautocreate=2
> ```

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 *Helix Core 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 Helix server license. A Helix server 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, Helix server displays an error message to that effect.

Deleting a user frees a Helix server 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 Helix server 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

Helix server 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 **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 Helix 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 files versioned by Helix server 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 Helix server 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 272.
- 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 199.
- 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" below.

Reclaiming disk space by archiving files

Over time, Helix 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 from 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

A version management system maintains the history (metadata) of what operations were performed on which files.

If certain files and their history are no longer in use, you might want to recover disk space on the central server. You can do one of the following:

- Move them into an archive depot by using the `p4 archive` command. If you later decide you want to use those file and recover their history, use the `p4 restore` command.
- Permanently delete the files by using the `p4 obliterate` command. One use case is to eliminate a submit or a branch that was created by mistake. However, the operations of **`p4 obliterate`** are computationally expensive. Avoid using **`p4 obliterate`** during peak usage periods because a large amount of metadata must be processed.
- Permanently delete the files but retain the metadata by using the `p4 obliterate` **`-p`** command. This option recovers the disk space of the files, yet reduces the amount of time the command runs because metadata is not processed.

Warning
Use **`p4 obliterate`** with caution. This is the one of only two commands in Helix server that actually remove file data. (The other command that removes file data is the archive-purging option for **`p4 archive`**.)

Warning
Do not use operating system commands (**`erase`**, **`rm`**, and their equivalents) to remove files from the Helix server root by hand.

By default, **`p4 obliterate`** ***`filename`*** does nothing. It merely reports on what it would do. To actually destroy the file and its metadata, 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.

Reclaiming disk space by removing orphaned archive files

You can save disk space by removing orphaned archive files and their associated metadata records. By "orphan", we mean a file that exists in the server depot's archive storage but that lacks a reference. Archives with a reference count of 0 are rare, but can be created by incomplete p4 submit or p4 shelve commands, system crashes, or other similar errors.

To reclaim that disk space:

1. Run `p4 storage -l` because it spawns a background process that looks for files in the specified ***//depotdirectory*** and its subdirectories that match the naming convention for the server's archive file format. Each file that is found along with its revision is checked for a matching entry in the `db.storage` table. If no such entry is found, a new record is created that describes the file and revision combination with a reference count of `0`.
2. Run **`p4 storage -d`** to delete those `0` references records along with the associated archive file.

For more details, see p4 storage in *Helix Core P4 Command Reference*.

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 Helix server superuser can use the commands **`p4 monitor pause`** and **`p4 monitor resume`**. If a process on a Helix Core server machine consumes excessive resources, it can also be marked for termination with **`p4 monitor terminate`**.

Once marked for termination, the process is terminated by the Helix 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 Helix server commands are running), entries may remain in the process table even after the process has terminated.

Helix server superusers can remove these erroneous entries from the process table altogether with `p4 monitor` **`clear`** ***`pid`***, where ***`pid`*** 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.

Terminating blocked processes

As soon as you are done "Enabling process monitoring" on page 201, each process that is added to the monitor table is eligible for a termination request. You can issue the following command:

```
p4 monitor terminate pid
```

where *pid* is a process identifier.

You then wait the full `db.monitor.interval`, and if the process was blocked waiting for client input, you can confirm that the Helix server terminated the process by looking at the output of **`p4 monitor show -ael`**.

Managing the database tables

Use the **`p4 dbstat`** command to display statistics on the internal state of the Helix server 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 Helix server installer supports scripted installation, enabling you to accelerate a deployment of Helix server across a large number of desktops.

Scripted installations are controlled by a configuration file that comes with the scriptable version of the installer for the Helix Core server. You can edit this file:

- to preconfigure Helix server environment variables (such as `P4PORT`) for your environment
- to automatically select Helix server applications in use at your site
- and more

For command-line options for automated deployment of Helix Client applications, see the Support Knowledgebase article, "Automated Deployment of Perforce P4V".

Troubleshooting Windows installations

Resolving Windows-related instabilities

Many large sites run a Helix server on Windows without incident. There are also sites in which a Perforce service or Helix 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 Helix server failures are caused by OS-level problems, a number of symptoms can indicate the OS is at fault. Examples include: the system crashing, the Helix Core 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 Helix server Command-Line Client (`p4.exe`) if they use the `P4EDITOR` or `P4DIFF` environment variables.

The reason for this is that Helix server 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 Helix server 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`. Helix server 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 Helix server 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.

Tuning Helix server for performance

This chapter explains factors that can affect the performance of Helix server, provides tips on diagnosing network-related difficulties, and offers suggestions on decreasing server load for larger installations.

Location of db.* files, journal, and depot files

We recommend separate filesystems for each of the following: db.* files, journal, and archive files.

	db.* files	Journal file	Depot files
Drive	For I/O requests that must be satisfied from beyond the filesystem cache, we recommend that: - The storage subsystem containing the `db.*` files have a memory cache, and that it be maximized. - Write-back caching be enabled, which 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. This might require direct connections between the host and the storage subsystem, and usually requires SSD or drives with the fastest rotational speed (such as 15K RPM). **Tip** Concerning solid state drives (SSDs), see "Disk Performance" in the Knowledge Base article, "Recommended Server Hardware Configurations".	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 - a logical drive and filesystem that is optimized for sequential writes.	The depot files should be located on a logical drive that is separate from: - the drive where the `db.*` files are located - the drive where the live journal is located. For best performance, the logical drive where the depot files are located should be on a separate storage subsystem connected to a separate host adapter.

	db.* files	Journal file	Depot files
Optimization	The `db.*` files need to be on a filesystem optimized for fast random read and write performance.	The live journal needs to be on a filesystem optimized for fast sequential write performance.	
RAID	RAID 1+0 (or RAID 10) is recommended for the logical drive where the `db.*` files are located. Generally, `p4d` 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 optimal stripe size might depend upon the number of physical drives in the logical drive. Hardware-based RAID implementations usually have good performance characteristics. Software-based RAID implementations can require CPU cycles that could otherwise be used for `p4d` processes.		The depot files require more disk space than the db.* files. The I/O throughput for depot files is not as critical as for the db.* files. Therefore, consider using an economical RAID configuration, such as RAID 5.

Hardware and performance

In general, Helix server performs well on any server-class hardware platform. We recommend using the latest Linux distributions. The following variables can affect the performance of Helix server.

Filesystems

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 Helix server 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 Helix server workloads.

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.

> **Tip**
> Increasing the read-ahead size might result in better performance for operations requiring sequential reads. However, during random reads, this might discard previously-cached data that might have satisfied subsequent requests.

CPU

CPU resource consumption can increase due to factors such as:

- complexity in the protections table
- compression and decompression
- lockless reads in some situations, although lockless reads are generally better for performance than file locking (see db.peeking)

When considering a CPU choice, we recommend the maximum clock rate that is available on the number of cores required for your typical workload. For detailed guidance for your typical workload, contact Helix Core Consulting.

Faster processors and memory in the machine where `p4d` executes might result in faster execution of `p4d` commands. Because 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. Some `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. Given the list of "Commands implementing lockless reads" on page 255, speeding commands through the server with a fast CPU might not be as critical from a concurrency point of view. Many commands can run concurrently through the Helix Core server, so 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 fast processors might need more processors or processors with more cores.

> **Note**
> If you are using SSL to secure client-server connections, choose a CPU that supports the AES instruction set. Helix server normally uses AES-256 to encrypt its SSL connections, so using a CPU that supports AES will minimize the encryption overhead.

Memory

Server performance is highly dependent upon having sufficient memory. 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.

Ensure that:

- the server doesn’t page or swap when it runs large queries
- the often-used pages from the `db.*` files can be cached in memory

Although most operations involve only a subset of files, multiple large operations can be performed simultaneously, and thus might require more memory to avoid paging.

Assuming you follow the best practice of having a device that contains *only* the database files, one way to determine if you have allocated sufficient memory is to verify that the physical read rate on this device is minimal.

Network

Helix server can run over any TCP/IP network. For remote users or multi-server 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.

Helix server 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 the TCP `TIME_WAIT` state, typically for two minutes. The port number ranges from `1025` to `32767` (or larger). Therefore, it is possible to occupy all available ports by invoking a Helix server command many times in rapid succession, such as with a script.

TCP keepalive

By default, keepalives are enabled if that functionality is supported by the OS. If your network silently drops idle connections, users might experience unexpected connectivity issues. The following p4 server configurables override the behavior configured in the operating system:

- net.keepalive.count
- net.keepalive.disable

- net.keepalive.idle
- net.keepalive.interval

For a general explanation of keepalive technology, see:

http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/overview.html

http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/usingkeepalive.html

Usage and performance

This section contains the following topics.

Use patterns

Helix server usage can affect performance. Usage patterns can have a direct effect on performance:

- The depot filenames are the leading portion of the key in several important `db.*` files, such as `db.rev`, `db.revhx`, and `db.integed`. As the length of paths to depot filenames increase, performance decreases. We recommend short names when possible.
- Rather than frequent creation of full branches:
  - Consider task streams. A task stream is a lightweight short-lived stream that only promotes edited files to the repository. See p4 stream.
  - If you are not using task streams, consider branching only the subset of files needed for a given bug fix.
  - Consider fixing multiple bug fixes on a single branch.

> **Tip**
> The frequent creation of full branches increases the amount of metadata. This results in more levels within the `db.*` file B-trees, more key comparisons, and more I/O requests to traverse to the leaf pages. Also, reading and writing large amounts of metadata might affect the filesystem cache to the detriment of other Helix server tasks.

Using read-only and partitioned clients in automated builds

Build automation scripts frequently create, sync, and tear down clients. This can fragment the db.have table, which can cause your end-users to experience slower sync operations. To avoid this problem, consider adding one or more additional client workspaces of type **readonly** or **partitioned** for the exclusive use of automated builds.

Readonly and Partitioned Clients

Client workspaces of type **readonly** and **partitioned** have their own **db.have** database table that is not journaled and is easily removed when the client workspace is deleted, thus preventing fragmentation of the **db.have** table. These types of client workspaces are ideal for the build automation cycle that creates a client workspace, syncs it with the depot, performs builds, and finally deletes the client workspace.

Use **readonly** client workspaces for short lived client workspaces used for build automation where editing and submitting files is not required. Typical commands run from a readonly client workspace:

- p4 clean
- p4 flush
- p4 status
- p4 sync
- p4 update

Use **partitioned** client workspaces for the same purpose as **readonly** client workspaces, but with the additional ability to edit and submit files. Typical commands run from a partitioned client workspace, in addition to the the **readonly** commands above:

- p4 add
- p4 edit
- p4 integ
- p4 reconcile
- p4 shelve
- p4 submit
- p4 unshelve

Creating a readonly or a partitioned client workspace

Before creating a **readonly** client workspace or a **partitioned** client workspace, configure the storage location for the **db.have** tables of these types of client workspaces by setting the client.readonly.dir server configurable. For example:

```
p4 configure set client.readonly.dir=part-db-have
```

> **Note**
> Although the name of this configurable contains "readonly", its setting also applies to partitioned client workspaces.

Relative paths specified in `client.readonly.dir` are relative to P4ROOT, but absolute paths can also be specified. The client.readonly.dir server configurable does not require a server restart. The Helix Server creates the directory upon first usage, if it doesn't already exist.

To create a client workspace that is `readonly` or `partitioned`, set the `Type:` field in the client specification to either `readonly` or `partitioned.`

For example,

```
p4 client my-readonly-client
...
Type: readonly
```

or

```
p4 client my-partitioned-client
...
Type: partitioned
```

> **Tip**
> Although a `readonly` client cannot be changed into a `writeable` client, a `readonly` client can be converted to a `partitioned` client by updating the `Type:` field in the client specification. See p4 client > Options > `-T type` in *Helix Core P4 Command Reference*.

Current Limitations

- Can't use `p4 changes -m1 @`***`partitioned-client-name`***
- Can't use `p4 fstat #have` from a `partitioned` client against a forwarding replica
- Can't use p4 switch from a `partitioned` client
- `p4 sync @partitioned-client-name` is equivalent to `sync #none`
- The db.working and `db.have` check performed by `p4d -xx` ignores the `db.have` tables for partitioned client workspaces. This results in the creation of `jnl.fix` records to delete the partitioned client's `db.working` records for any open file revisions. Before replaying a `jnl.fix` file, contact Technical Support.

Using parallel processing for submits, syncs, and shelves

Submits and syncs

You can configure the server to allow parallel file transfer for submit and sync processing. Parallel processing is most effective with long-haul, high-latency networks, or with other network configurations that prevent the use of available bandwidth with a single TCP flow. In addition, parallel processing between a client running on a multi-core machine and the server allows the uncompression of large compressed binary files to happen in parallel on the client.

- Use the **`net.parallel.max`** configurable to:
  - Transfer files in parallel during the submit process.
  - 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.
- Use the **`net.parallel.submit.threads`** configurable to specify the number of threads to be used for sending files in parallel for each submit (P4V 2017.3 and later).
- Use the **`net.parallel.threads`** configurable to turn on parallel sync in a server. This parameter specifies the number of independent network connections that can be used for syncing files concurrently for each sync. When this parameter is set, parallel sync is automatically enabled in P4V as well (P4V 2017.3 and later).
- To reduce lock contention during parallel syncs, set the `client.sendq.dir` configurable.

For more information, see the p4 submit and p4 sync commands in *Helix Core P4 Command Reference*.

Shelves

A p4 shelve command might execute more rapidly by transferring multiple files in parallel.

To enable automatic parallel shelving, set the net.parallel.shelve.threads configurable to a value that is less than or equal to the value of the **`net.parallel.max`** configurable.

Optionally, set the net.parallel.shelve.min configurable to change the minimum number of files in a parallel shelve, which, by default, is 9.

Optionally, set the net.parallel.shelve.batch configurable to change the number of files in a parallel shelve, which, by default, is 8.

To disable automatic parallel shelving, unset the **`net.parallel.shelve.threads`** configurable.

Note
Promoted shelves require an additional file transfer from the Edge to Commit Server. Parallel pull threads for this transfer are only used if the ExternalAddress field is set in its Edge Server spec and pull threads can be used on the Commit Server. This transfer using pull threads is currently not supported on Windows platforms.

A user can override the shelve configurables on the command line, or disable parallel shelving with the `p4 shelve --parallel=0` command.

Improving concurrency with lockless reads (peeks)

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.

To change the setting of lockless reads on your Helix Core server, use the `p4 configure set db.peeking=N` command.

Tip
`db.peeking` is a dynamic configurable, but prior to the 2017.1 release, a change to the value of this configurable required a server restart.

Possible values for `db.peeking`

Value	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 Helix server 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. This configuration is expected to provide the best performance results for most sites. It is the default value.

Value	Meaning
`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`	
`dirs`	
`filelog`	
`files`	when `db.peeking=3`
`fixes`	
`fstat`	when `db.peeking=3`
`have`	

Command	Notes
interchanges	
integrate	
integrated	
istat	
jobs	
keys	
labels	
merge	
print	Applies to `print -a`
resolved	
sizes	Applies to `sizes -a`
streams	
sync	when `db.peeking=3`
users	
verify	

In most cases the following commands 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 p4 fstat:

```
$ p4 -Ztrack -Zpeeking=1 fstat <a_file>
```

```
...
— db.working
— pages in+out+cached 3+0+2
— locks read/write 1/0 rows get+pos+scan put+del 0+1+1 0+0
...
```

and compare the results with:

```
$ p4 -Ztrack -Zpeeking=2 fstat <a_file>
```

```
...
— db.working
— pages in+out+cached 3+0+2
— locks read/write 0/0 rows get+pos+scan put+del 0+1+1 0+0
— peek count 1 wait+held total/max 0ms+0ms/0ms+0ms
...
```

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 Helix server 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 (**`db.peeking`** is set to **`0`** or unset), the attempt is ignored.

However, this protection is not available with the "side-track servers" described in the Support Knowledgebase article,"Setting Up a 'Side-track' Server to Control Priority".

> **Warning**
> All side-track servers must have the same **`db.peeking`** setting as the main server or server deadlock might occur.

Diagnosing slow response times

Helix server is normally a light user of network resources. Although it is possible that an extremely large user operation could cause the Helix 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 Helix Core 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, the performance of Helix server 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 are not optimal. Users and administrators can ease load on their servers:

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 Helix server 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 270 and "Using a temporary client workspace" on page 271) 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 147) are actually another type of Helix server 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 Helix server 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 Helix server, a Helix server 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/...
```

Helix server 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 Helix server 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 **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 cause difficulties for your users, do not use them unless you find that certain operations are slowing down your server. Because some Helix server applications can perform large operations, you should typically set:

- `MaxResults` no smaller than 10,000

  `MaxScanRows` no smaller than 50,000

  `MaxLockTime` within the 1,000-30,000 (1-30 second) range.

For more information, including a comparison of Helix server commands and the number of files they affect, use the command-line help:

```
$ p4 help maxresults
$ p4 help maxscanrows
$ p4 help maxlocktime
```

MaxResults, MaxScanRows and MaxLockTime for users in multiple groups

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`, that user is still limited by the highest numeric `MaxResults` (or `MaxScanRows` or `MaxLockTime`) setting of the other groups of which this user is a member.

A user’s commands are 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`.

Note
Ticket **`Timeout`** and **`PasswordTimeout`** values for users who belong to multiple groups are calculated to be the largest timeout value for all the groups of which the user is a member, including **`unlimited`**, but ignoring **`unset`**.

A user who is not a member of any group has the default ticket **`Timeout`** value of **`43200`** seconds, which equates to twelve hours, and the **`PasswordTimeout`** value of **`unset`**.

To create a ticket that does not expire, set the ticket **`Timeout`**value 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 contain lines of the form:

```
Server is now using nnn active threads.
```

Tip
You can use the server log to determine what levels of activity are typical for your site. Set `server.maxcommands` to a value higher than your anticipated peak activity. We recommend at least 200% to 500% higher.

Too many commands

Starting in 2019.1, if a problem occurs with `server.maxcommands`, users who are super or **`operator`** (see "User types" on page 231 can still run commands to diagnose the problem.

Even if end-users see the **`TooManyCommands`** error, a user who is **`super`** can still run the following commands:

p4 admin	p4 configure	p4 counter
p4 counters	p4 group	p4 groups
p4 help	p4 help-graph (graph)	p4 info
p4 journals	p4 lockstat	p4 login
p4 login2	p4 logout	p4 logtail
p4 monitor	p4 protect	p4 protects

p4 pull	p4 server	p4 servers
p4 user		

and a user who is `operator`, such as the administrator of an edge server, can still run the following commands:

p4 admin	p4 configure	p4 counter
p4 counters		
		p4 info
	p4 lockstat	p4 login
	p4 logout	p4 logtail
p4 monitor	p4 protect	
p4 pull		p4 servers
p4 user		

Side-tracking builds to prioritize interactive commands

During peak hours at large sites, the combination of high command volume and commands run against large data sets can increase the load on server memory. This might cause paging at the server machine and slow command response at the client machine.

To give priority to interactive user commands on the main server port, you can set up a separate port on the same machine that handles low priority commands one at a time. This reduces the load of concurrent commands on the main server port.

Thus a single server with two ports acts almost as if it were two servers, the main server for immediate interaction, and the "side-track" server for builds or batches where a slower command response might not matter.

> **Tip**
> Do not expect the side-track server to provide any benefit for single commands because it encounters the same bottlenecks as the main server when locking the database files and writing to the journal.

Side-track server port

The side-track server:

- listens on a P4PORT that is separate from the main `P4PORT`.
- handles each request one-by-one

Example of two ports on one server

Suppose the main Perforce server runs on `computer:1666` as its `P4PORT`.

On the same machine, you can then start up a side-track server that uses `computer:1667` as its `P4PORT` by issuing the command:

```
p4d -p computer:1667 -f
```

The `-f` flag causes the side-track server to run as a single-threaded process. See "Server options" on page 500.

Make sure that all the other server configuration variables are set the same as the main server settings, or include them on the p4d command line.

Using for batches and builds

Users who want to run a large command, such as a large build, through the side-track server can set their `P4PORT` to `computer:1667` or use the `-p computer:1667` option on the p4 command line. Batch scripts can be configured to run with `P4PORT=computer:1667` so they avoid the main server.

The side-track server operates against the main Perforce database files. In other words, the side-track server shares the same Helix serverP4ROOT and the same journal file.

The side-track server get its ` P4ROOT` and `P4JOURNAL` environment variable values from the main server. Log entries for it can be sent to either the main server log (P4LOG) or to a separate log file, by specifying it in the side-track server command invocation. For example:

```
p4d -p computer:1667 -f -L p4sidetrack.log
```

On the client side, a user can redirect requests on-the-fly to the side-track server by using the global `-p` option with `p4` commands to change the server port setting for that command. For example:

```
p4 -p computer:1667 diff -se
```

For details, see Global options in *Helix Core P4 Command Reference*.

Using for diagnostic logging

Using a separate log file for recording errors from the side-track server can be useful for diagnostic purposes. This is a way to avoid causing the main server log file to grow excessively.

For example, detailed logging of RPC messages can grow large. It might be useful to look at the client/server protocol output using the `rpc=3` flag to see how communication is being interrupted. The debugging flag output that goes to STDOUT can be redirected to a file:

```
p4d -v rpc=3 -p 1667 -L p4sidetrack.log 1 > p4debug.1667
```

The side-tracker server can aid diagnostics by having all the activity it logs be on a single thread. However, if you have a non-SSL server and an SSL both running, you do not have a single thread.

Unloading infrequently-used metadata

Over time, Helix 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.

> **Note**
> The output of `p4 -ztag info` includes `unloadSupport enabled` if the administrator has created a depot of type `unload`. If not, the output includes `unloadSupport disabled`.

Unload old client workspaces, labels, and task streams

The `p4 unload` command transfers infrequently-used metadata from the Helix Core server `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 264) that users do not need to ask for assistance with **`p4 unload`** or **`p4 reload`** operations.

Accessing unloaded data

By default, Helix server 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 *Helix Core 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 Helix server Command-Line Client, **`p4`**, supports the scripting of any command that can be run interactively. Helix 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 Helix server command issued causes a connection thread to be created and a **`p4d`** subprocess to be started. Reducing the number of Helix server 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 Helix server 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 Helix server 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 Helix Core 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 Helix Core 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 268.

Using a temporary client workspace

Most Helix server 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 Helix Core 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 Helix Core 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 76.
- By default compression is enabled between the Helix Core 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 Helix Core 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

The internal database stores its data in structures called 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 Helix server checkpoint and restore processes (see "Backup and recovery concepts" on page 176) 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 Helix server 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 Helix server 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 the topic on upgrading "From prior to 2013.3" on page 67.)

Customizing Helix server: job specifications

The Helix server jobs feature enables users to link changelists to enhancement requests, problem reports, and other user-defined tasks. Helix server also offers P4DTG (Helix Defect Tracking Gateway) as a means to integrate third-party defect tracking tools with Helix server. See "Working with third-party defect tracking systems" on page 283 for details.

The Helix server user's use of the `p4 job` command is discussed in the *Helix Core Server User Guide*. This chapter covers administrator modification of the jobs system.

The default jobs template has five fields for tracking jobs. These fields are sufficient for small-scale operations, but as projects managed by Helix Core server grow, the information stored in these fields might be insufficient. To modify the job template, use the `p4 jobspec` command. You must be a Helix Core server administrator to use **`p4 jobspec`**.

This chapter explains how to modify the Helix server job template.

> **Warning**
> Improper modifications to the Helix server 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 Helix server job template

To understand how Helix server jobs are specified, consider the default Helix server job template. The examples that follow in this chapter are based on modifications to the this template.

A job created with the default Helix server 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 Helix server jobs stored on your server.

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 Helix server.
`Comments:`	The comments that appear at the top of the `p4 job` form. They are also used by P4V, the Helix 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 Helix server 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 Helix 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 Helix server 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. ■ `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 Helix server 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
`$user`	The Helix server user creating the job, as specified by the `P4USER` environment variable, or as overridden with `p4 -u` ***`username`*** `job`.
`$now`	The date and time at the moment the job is saved.

Variable	Value
`$blank`	The text **`<enter description here>`**. When users enter jobs, any fields in your jobspec with a preset of **`$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`**.

If **`completed`** is in a **`select`** setting in the **`Values:`** field of p4 jobspec, and if we want the default fix status to be **`completed`**, we can use **`/completed`** as follows:

```
Presets:
        Status open,fix/completed
```

If we wanted 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, we could use **`/same`** as follows:

```
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 Helix 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 Helix server 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 Helix server and cannot be renamed nor deleted.

  Fields 102 through 105 are reserved for use by Helix server applications. Although it is possible to rename or delete these fields, it is highly undesirable to do so. Helix server 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 Helix 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/unknown
Presets:
        Status open
        User $user
        Date $now
        Type unknown
        Priority unknown
        Subsystem unknown
        Owned_by $user
        Description $blank
Comments:
        # Custom Job fields:
        # Job:    The job name. 'new' generates a sequenced job
number.
        # Status: '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:
        #                 'bug', 'sir', 'problem' or 'unknown'
        # Priority:       How soon should this job be fixed?
        #                 The value is 'a', 'b', 'c', or 'unknown'
        # Subsystem:      The value is 'server', 'gui', 'doc', 'mac',
or 'unknown'
        # Owned_by:       Who's fixing the bug. Can be changed.
        # 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. Therefore, fields do not need not be ordered by numeric identifiers.

If the user named **`marie`** runs **`p4 job`** against the example custom jobspec, it displays the following job form:

```
# Custom Job fields:
# Job:    The job name. 'new' generates a sequenced job number.
# Status: '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:
#                 'bug', 'sir', 'problem' or 'unknown'
# Priority:       How soon should this job be fixed?
#                 The value is 'a', 'b', 'c', or 'unknown'
# Subsystem:      The value is 'server', 'gui', 'doc', 'mac', or
'unknown'
# Owned_by:       Who's fixing the bug. Can be changed.
# Description: Comments about the job. Required.
Job:    new
Status: open
User:   marie
Type:   unknown
Priority:       unknown
Subsystem:      unknown
Owned_by:       marie
Description:
        <enter description here>
```

Working with third-party defect tracking systems

Perforce currently offers two independent platforms to integrate Helix server with third-party defect tracking systems. Both platforms allow information to be shared between Helix server's job system and external defect tracking systems.

P4DTG, the Helix Defect Tracking Gateway

P4DTG, the Helix 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 Helix server 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:

https://www.perforce.com/plugins-integrations/defect-tracking-gateway

Available from this page are an overview of P4DTG's capabilities, the download for P4DTG itself, and a link to the *Helix Defect Tracking Gateway Guide*, which describes how to install and configure the gateway to replicate data between a Helix Core server and a defect tracker.

Building your own integration

Even if you don't use Helix server integrations as your starting point, you can still use the job system as the interface between Helix server 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 Helix server 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 Helix server with the `-i` option is available in the *Helix Core P4 Command Reference*.

- A trigger on the Helix server side that checks changelists being submitted for any necessary bug fix information.

For more about triggers, including examples, see "Triggers and Extensions" on page 284.

Triggers and Extensions

Triggers and server extensions allow you to customize parts of the Helix Core with user-supplied logic.

	Server Extensions	Triggers
Release	New in 2019.1	Have been a feature of Helix Core server for twenty years
Status	Supported as of 2019.1	Remain fully supported
Environment	Part of the Helix core product executables. No external dependencies. Written in the Lua programming language. Not platform-specific.	An external program supplied and managed by the Helix core administrator.
Starting point for command-line help	command-line: `p4 help extension`	command-line: `p4 help triggers`

To learn about and begin using server extensions, see the *Helix Core Extensions Developer Guide*.

Triggers

Triggers

Helix server supports **triggers**, which are user-written programs or scripts that are called when certain operations are performed. Examples of operations that might fire a trigger are changelist submits, changes to forms, and attempts by users to log in or change passwords.

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 functionality. Consider the following common uses:

- To validate changelist contents beyond the mechanisms afforded by the 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 command.
- To validate forms, or to provide customized versions of 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 Helix server 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 "Authentication options" on page 129.
- To retrieve content from data sources archived outside of the repository.

Important
Be aware that the client's settings might require adjustment. For example, to see the server's output, you might need to enable logging on the P4V client. See the Knowledge Base article, "Debugging 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.

Note
See also "Triggers and commit-edge" on page 451 in "Managing commit-edge installations" on page 447.

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 Helix server 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 Helix server, 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 352.

   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 292.

   Triggers can also use the command-line client (**`p4.exe`**) or the Helix server scripting APIs (P4Ruby, P4Python, P4PHP) when data is needed that cannot be accessed by trigger variables. For more information, see *Helix Core 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 295.

   Triggers can be written for portability across servers. For more information, see "Writing triggers to support multiple Helix servers" on page 297.

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 288.

> **Warning**
> When you use trigger scripts, remember that Helix server 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 Helix server 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 Helix server 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.

Field	Meaning
`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 Helix server to run when the conditions implied by the trigger definition are 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 Helix 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 Helix server 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"` 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 295 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=abR)aCad^ab9ra
cd /home/perforce/my-database-triggers

p4 admin checkpoint
ls -l checkpoint.* journal*
```

where **/home/perforce/my-database-triggers** represents the location of your triggers.

We recommend 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, Helix server 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 level 3 of "Server security levels" on page 130). This prevents storing a plaintext password value in P4PASSWD. Instead, set **P4PASSWD** to the ticket value that **p4 login -p** returns.
- 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.
- Helix server commands in trigger scripts are always run by a specific Helix server user. If no user is specified, an extra Helix server 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 Helix server 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 Helix Core 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 Helix server-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 the facing page describes how to select the method used for communication and how to parse dictionary input.
- "Storing triggers in the depot" on page 295 describes how to format depot paths if you want to run a trigger from the depot.

- "Using multiple triggers" on page 296 explains how Helix server interprets and processes the trigger table when it includes multiple trigger definitions.
- "Writing triggers to support multiple Helix servers" on page 297 describes how you can write a trigger so that it is portable across Helix server installations.
- "Triggers and multi-server architecture" on page 298 explains the issues you must address when locating triggers on replicas.

For information about debugging triggers, see the Support Knowledgebase article, Debugging Triggers,

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 352 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 should exit with a zero value.	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
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 352 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 Helix server 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.

Example **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 **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 Helix server database)
4. `form-delete` (validated form is already stored in the Helix server database)

Exclusionary mappings for triggers

Example

```
trig1 change-submit //depot/... "trig.pl %changelist%"
trig1 change-submit -//depot/products/doc/... "trig.pl %changelist%"
```

Submitting a change in `//depot/products/doc/...` results in the `/usr/bin/trig.pl` script NOT running.

Submitting a change in any other directory runs the first instance of a `trig1` script, that is, the script on the first `trig1` line and ignores the second instance of `usr/bin/trig.pl`.

Rules for exclusionary mappings

1. Exclusions must be LAST.
2. The same script or action must be associated with each different line of the same named trigger. When the path or file check falls through to a triggerable path or file, the script or action runs that is associated with the FIRST trigger line.
3. If you want a submit to fail, associate an exit(1) return code with the successful match of the path or file.

Writing triggers to support multiple Helix servers

To call the same trigger script from more than one Helix Core 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:

```
#!/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 352.

Triggers and multi-server architecture

Triggers installed on the master server must also exist on 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 by storing the trigger script 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. See "Storing triggers in the depot" on page 295.

Triggers installed on the replicas must have the same execution environment for the triggers and the trigger bodies. This typically include trigger login tickets or trigger script runtimes, such as Perl or Python.

> **Note**
> Edge servers have triggers that fire between client and edge server, and between edge server and commit server. See "Triggers and commit-edge" on page 451.

Defining background tasks in the triggers table

The Helix Server can use the triggers table to define background tasks that can be run with the `p4 bgtask` **`-t`** ***`bgtask-name`*** command or as a task at start up.

Example background task definition in the triggers table:

```
log_checker bgtask unset "p4 -p popeye:18100 logstat -s"
```

The defined **`log_checker`** bgtask can be:

- invoked at the command-line by using `p4 bgtask -t log_checker` or
- configured as a task to be run at server startup with `p4 configure set "startup.1=bgtask -t log_checker"`

The following table describes the fields of an archive trigger definition:

Field	Meaning
`name`	The name of the task, such as `log_checker`
`type`	`bgtask`
`path`	Use `unset` as the path value
`command`	The command or script the server will run in the background

Triggering on submit and populate

To configure Helix server 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.

> **Note**
> The p4 populate command branches a set of files (the source) into another depot location (the target) in a single step. Therefore, to access the files of a change being submitted by p4 populate, use a change-content trigger (see "Change-content triggers" on page 302) and the `p4 files @=change` command. (In the case of `p4 populate`, a change-content trigger is equivalent to a change-submit trigger because no file transfer takes place from the client.)
>
> "Change-submit triggers" on page 301 can use the `%command%` trigger script variable to ignore populate commands (`%command% == user-populate`) so that change-content triggers can process `p4 populate` commands.

The following table describes the fields of a submit trigger. For sample definitions, see the subsequent sections, describing each trigger subtype.

Field	Meaning
`name`	The name of the submit trigger.

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: ■ 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.

Field	Meaning
command	The trigger for Helix 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 Helix 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 Helix server 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 295 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

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.

Tip
Replicas that are metadata-only do not support triggers of type `change-commit` and `change-content`.

Example

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.

Tip
Replicas that are metadata-only do not support triggers of type `change-commit` and `change-content`.

Example

Here is 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

Note

p4 push and p4 fetch are commands specific to the Perforce proprietary distributed version control system (DVCS). See Using Helix Server for Distributed Versioning.

There is no requirement that any triggers be run at any point in the submission or push process.

To configure Helix server to run trigger scripts when the **`p4 push`**, **`p4 unzip`**, or **`p4 fetch`** commands are invoked, use *push 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 for Perforce DVCS.

For a description of the triggers that can be used by the server receiving the pushed items or responding to the fetch request, see "Additional triggers for push and fetch commands" on page 316.

Similarity between p4 submit and p4 push

During a push, the local server acts as the client of the shared server. Therefore, there are similarities between submits and 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.

The following figure:

- illustrates the similarities between submits and pushes
- illustrates the path of submitted files, via a changelist, from the client, to the local server, and finally, to the shared server
- includes all possible types of triggers and shows the types of triggers that can be run during each phase of these processes.

Figure 13-1 Change and push triggers

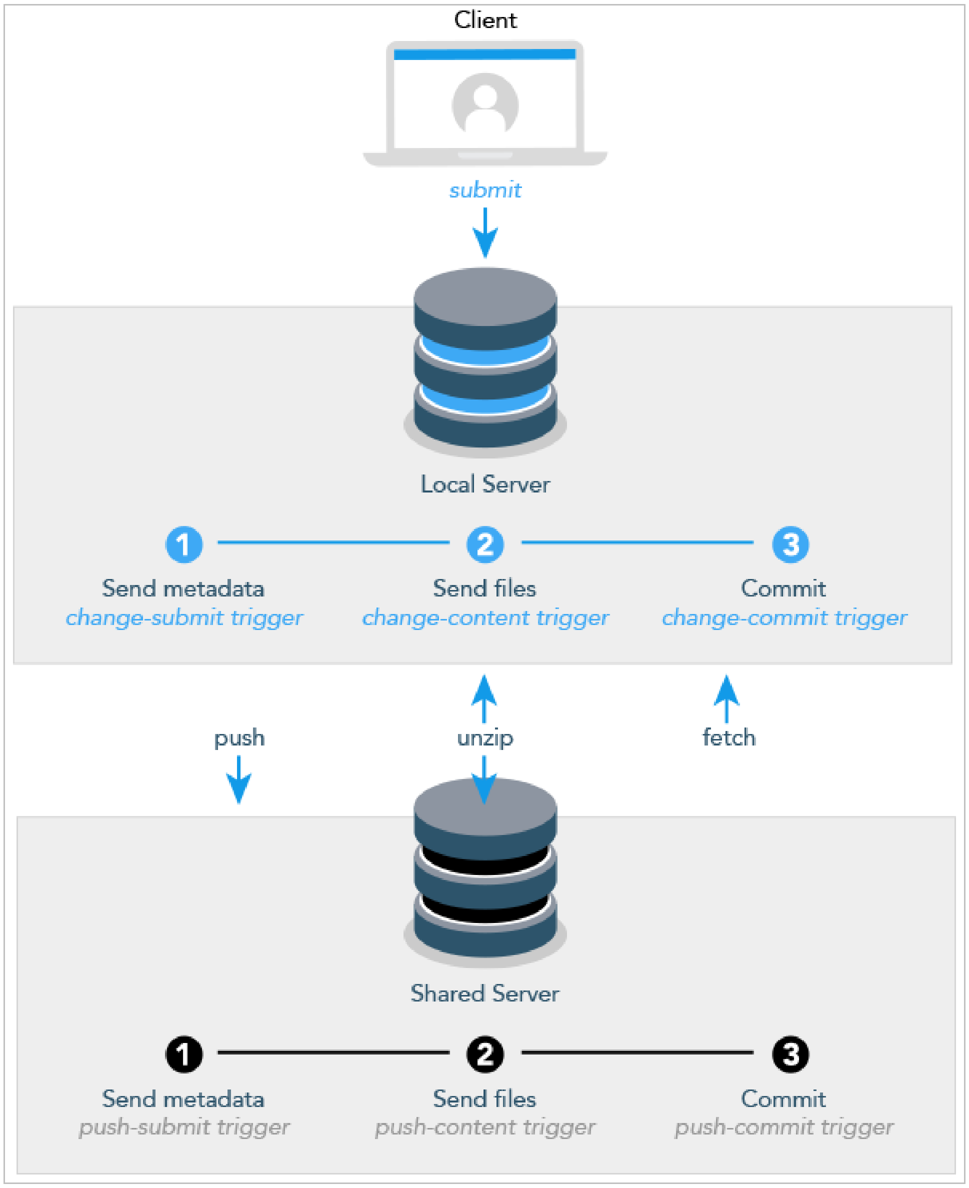


Figure 13-2

The three phases of submits and pushes include the following:

1. **Send metadata** causes metadata to be 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. **Send files**. The 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. **Commit**. 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.

Differences between p4 submit and p4 push

Differences between submits and pushes:

- While both submits and pushes are atomic
  - a submit encompasses a single changelist
  - push can contain multiple changelists. Thus the failure of a push is more costly.
- Submits are unidirectional.
- Pushes are the result of a `p4 push`, `p4 fetch`, or `p4 unzip` and 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 can access. Each `push-commit` trigger is a separate process with its own memory. See "Push-submit triggers" on page 310.
- 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.

Fields on a p4 push trigger

The following table describes the fields of a push trigger. For sample definitions, see the subsequent sections, describing each push trigger type.

Field	Meaning
`name`	The name of the push trigger.

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.
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 Helix 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 Helix 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 Helix server 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 295 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 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

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 approved ("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

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

Following is 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 0Fo
```

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" below 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 Helix server to run a trigger before or after a given command executes. You might want to execute a script:

- before a command runs to prevent that command from running
- after a command has run if you want to connect its action with that of another tool or process

> **Note**
> If you are Using Helix Core Server for Distributed Versioning, consider "Triggering on pushes and fetches" on page 305.

The following table describes the fields of the `command` trigger.

Field	Meaning
name	The name of the command trigger.
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 the grammar of regular expressions, see `p4 help grep`. Examples of possible values: - `pre-user-login` - `post-user-add` - `post-user-edit` - `pre-user-obliterate` - `pre-user-sync` - see "Example command trigger" on the next page below - `post-user-sync` To match a command name that is a substring of another valid command, use the end-of-line meta-character to terminate matching. For example, use `change$` so you don't also match `changes`. You cannot create a `pre-user-info` trigger. See also "Additional triggers for push and fetch commands" on page 316.

Field	Meaning
`command`	The trigger for Helix server to run when the condition implied by *`path`* is satisfied. Specify the command in a way that allows Helix 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 Helix server 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 be `"/usr/bin/perl %//depot/scripts/myScript.pl%"`. See "Storing triggers in the depot" on page 295 for more information.

Example command trigger

You might want a `pre-user-sync` trigger to run before a user's request to sync (p4 sync) is executed.

An example of a trigger table entry, `check-sync command pre-user-sync "check-sync.pl %serverservices%"`, is shown below in a comment:

```
#!/usr/bin/perl
# Example pre-user-sync command trigger
# Trigger table entry:
# check-sync command pre-user-sync "check-sync.pl %serverservices%"

$serverservices = $ARGV[0]; # from %serverservices% in trigger
table
$allowed_to_sync_from = 'edge-server';
if ($serverservices eq $allowed_to_sync_from) {
    exit 0;
    } else {
    print 'Sync must only be run from edge servers !';
    exit 1;
}
```

Additional triggers for push and fetch commands

The section "Triggering on pushes and fetches" on page 305 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 responding (or remote "-rmt-") server can use these triggers:

Trigger	Run this trigger on the remote server
`pre-rmt-Push`	Before it receives pushed content
`post-rmt-Push`	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`	before it responds to a fetch request
`post-rmt-Fetch`	After it responds to a fetch request

Triggering on journal rotation

To configure Helix server 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
`name`	The name of the trigger.
`type`	- `journal-rotate`: Execute the trigger after the journal is rotated and database file locks are released. - `journal-rotate-lock`: Execute the trigger after the journal is rotated but while the database files are still locked. While the database tables are locked, no P4 commands can be run against this Helix server. **Warning** While a `journal-rotate-lock` trigger is running, the Helix server will not respond to Helix client commands.
`path`	The server(s) on which the triggers should be run. One of the following: `any`: all servers *`serverid`*: the specified server
`command`	The trigger for Helix server to run when the server matching *`path`* is found for the trigger type. Specify the command in a way that allows Helix 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 Helix server 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 295.

Triggering on shelving events

To configure Helix server 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
name	The name of the trigger.
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.
command	The trigger for Helix server to run when a matching ***path*** applies for the trigger type. Specify the command in a way that allows Helix 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 Helix server 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 295 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

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

Here is an example of a shelve-commit trigger that notifies users (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

Here is an example of 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 Helix server 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 305.

The following table describes the fields used for a fix trigger definition.

Field	Meaning
name	The name of the trigger.
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 Helix server to run when a user adds or deletes a fix. Specify the command in a way that allows Helix 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 Helix server 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 295 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

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 Helix server 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
`name`	The name of the trigger.

Field	Meaning
type	- **form-save**: Execute a form trigger after the form contents are parsed, but before the contents are stored in the Helix server 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 Helix 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 Helix server 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 Helix server to run when the type of form specified in the ***path*** field is processed. Specify the command in a way that allows Helix 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 Helix server 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 295 for more information. For **form-in**, **form-out**, **form-save**, and **form-delete** triggers, the data in the specification becomes part of the Helix server 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 Helix server metadata.

Example

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 Helix Core server generates a form for display to the user.

Warning
Never use a Helix server 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

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 Helix Core server.

Example

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 Helix Core server for parsing. Otherwise, an error message is displayed, and changes to the job are rejected.

Tip
For detailed guidance for using the flag for Python marshaled objects, see the Support Knowledgebase article, "Using p4 -G".

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 Helix server, but before the form is deleted from the Helix server database.

Example

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

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 Helix server 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.

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 Helix server's internal user database, and it is more secure. For more information, see "Authentication options" on page 129.

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 Helix server.

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 the Support Knowledgebase article, "Authenticating with LDAP".

You must restart the Helix Core 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 Helix server **`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
`name`	The name of the trigger.
`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 Helix Core 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 Helix Core server after adding a **`service-check`** trigger. See also the "Optional auth-invalidate trigger" on page 338.
`path`	Use **`auth`** as the path value.

Field	Meaning
command	The trigger for the Helix Core 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 Helix Core 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 Helix server 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 295 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 Helix server 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 Helix server administrators who want to use LDAP to authenticate other Helix server s in replicated and other multi-server environments.

> **Warning**
> If you are using **`auth-check`** triggers, the Helix server superuser must also be able to authenticate against the remote authentication database. (If you, as the Helix server 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 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 sign-on and auth-check-sso triggers

Client script and server script

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 Helix Core server, where the server-side script runs.

Client-side script	Server-side script
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 Helix server. **`P4LOGINSSO`** contains: ■ the name of the client-side script ■ zero or more of the following trigger variables, passed as parameters to the script: • **`%user%`** • **`%serverAddress%`** • **`%P4PORT%`** For example, `$ export P4LOGINSSO="/path/to/sso-client.sh %user% %serverAddress% %P4PORT%"` Where **`%user%`** is the Helix server client user, **`%serverAddress%`** is the address of the target Helix 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 multi-server configuration in which a proxy or broker acts as an intermediary between the client and the server: ■ the **`%serverAddress%`** variable holds the address/port of the server ■ the **`%P4PORT%`** variable holds the port of the intermediary. The script decides what to do with this information.

p4 login behavior with auth-check-sso trigger

The table below describes the behavior of p4 login when a trigger of type **`auth-check-sso`** is in place.

Depending on the configuration and environment:

- The P4LOGINSSO client-side script is executed, performing customizable SSO operations, potentially without user interaction.
- The user is prompted to authenticate by password. This is a fallback if no client-side **P4LOGINSSO** script is configured.
- The user's login attempt is rejected until a valid **P4LOGINSSO** script is configured in the user's environment.

Version 2018.2 introduced two configurables that change the default behavior:

- auth.sso.allow.passwd allows a user whose password is stored in the database (**db.user**) to fall back to password authentication if **P4LOGINSSO** is not configured.
- auth.sso.nonldap allows a user whose **AuthMethod** is set to **perforce** on an LDAP-enabled server to make use of **P4LOGINSSO**. See "Defining authentication for users" on page 133.

If P4LOGINSSO is set:

		Not LDAP-Enabled		**LDAP-Enabled**	
auth.sso.allow.passwd	**auth.sso.nonldap**	**db.user only**	**auth-check trigger**	**'perforce' + LDAP enabled**	**'ldap' + LDAP enabled**
0	0	**P4LOGINSSO fires**		Password requested	**P4LOGINSSO fires**
	1			**P4LOGINSSO fires**	
1	0			Password requested	
	1			**P4LOGINSSO fires**	

If P4LOGINSSO is not set:

		Not LDAP-Enabled		LDAP-Enabled	
auth.sso.allow.passwd	**auth.sso.nonldap**	**db.user only**	**auth-check trigger**	**'perforce' + LDAP enabled**	**'ldap' + LDAP enabled**
0	0	**Login rejected**	Password requested	Password requested	Password requested
	1			**Login rejected**	
1	0	Password requested	Password requested	Password requested	Password requested
	1				

Example Interaction between client-side and server-side scripts

> **Note**
> What follows is a trivial example. For a substantial example and a step by step walk-through for both Linux and Windows, see the Knowledge Base article, "Setting up Single Sign-On (P4LOGINSSO)".

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 Helix Servers.
```

```
#
# In this example, we simulate a client-side authentication process
# based on whether the user is connecting to the same Helix 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 Helix Core 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 Helix Core server (for example, by running **`p4 -p`** ***`host`*****`:`*****`port`*** **`login`** against a different Helix Core 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 Helix Core 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.

Optional auth-invalidate trigger

The optional **`auth-invalidate`** trigger type is fired when a user's ticket is explicitly invalidated by p4 logout. This trigger can propagate the logout to an authentication systems that is external to Helix Core. For example, the trigger can be used between the SAML SSO agent and the identity provider (IdP).

For an auth-invalidate trigger:

`%user%` -- the user's username

`%fullname%` -- the user's fullname

`%email%` -- the user's email address

`%host%` -- the host IP address of the ticket being invalidated or **`all-hosts`** for p4 logout **`-a`**

`%2fa%` -- **`true`** if the multi factor authentication status was reset, **`false`** if the user doesn't use multi factor authentication and **`only2fa`** for **`p4 logout -2`**

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 Helix Core server prompts the user to enter his or her old password.
3. The Helix Core 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 Helix Core 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 Helix server. The **`auth-set`** trigger type is included mainly for completeness.

Because the Helix Core 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. A trivial authentication-setting script

Example **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 for multi-factor authentication (MFA)

> **Tip**
> To get command-line help for MFA, type `p4 help mfa` to see the topic "MFA - Multi-Factor Authentication".

Support for multi-factor authentication is provided by installing three mandatory triggers:

- `auth-pre-2fa`, which presents the user a list of authentication methods
- `auth-init-2fa`, which begins the flow for the chosen authentication method
- `auth-check-2fa`, which checks whether passwords are valid

Only one trigger of each type can be defined and all three triggers must be present.

These triggers return JSON results to the server. Once installed, and the server has been restarted, the security level is set to `3` implicity, and can be explicitly set higher.

To configure a user to require MFA, the `AuthMethod` field in the user spec for that user must be modified to either `perforce+2fa` or `ldap+2fa`. This will require that this user run the p4 login2 command to perform the second authentication steps. If automatic login prompting is enabled, users will automatically perform this after their normal password based authentication. See the `p4 help login2` command-line help.

There are three phases to MFA, each based on the execution of that phase's trigger:

The list-methods phase (auth-pre-2fa)

This phase presents the user with a list of available MFA methods. For example, users might be configured to use either SMS or a mobile authentication application. In interactive mode, if the user only has one method, this is chosen automatically. These methods are returned by the `auth-pre-2fa` trigger. The trigger can also indicate that this user doesn't require additional authentication at this time, or that this user is not permitted access. This trigger is expected to return `0` on success and return a JSON string to the server via STDOUT.

The JSON response should be in the following format:

```
{
    "status" : 0,
    "methodlist" : [
        [ "method1" , "method description1" ],
        [ "method2" , "method description2" ]
    ],
    "message" : "Error message"
}
```

The status field is required, and should be be `0` on success with the `methodlist` populated with a dictionary of authentication, where the key is the method name and the value is the method description.

If the status is set to `2`, MFA is not required for this user on this host at this time. Any other status value is considered to be a rejection of the authentication attempt. In these cases, the methodlist is not needed and instead a message can be provided to be relayed to the user.

For example:

```
{
    "status" : 2,
    "message" : "Second factor authentication not required"
}
```

The init-auth phase (auth-init-2fa)

This phase begins the second authentication flow for the chosen method. It calls the `auth-init-2fa` trigger, returning the status (`0` for success) and the scheme. In the case of an error, the status would be non-zero and the scheme is not required. In addition, a message might be reported to the user in either case.

An optional challenge can be set to be presented to the user. For authentication flows that require state between init and check, a token can be set. This token is stored in the server but never reported to the user. The token is available to the next trigger via a trigger variable.

Here is an example JSON response:

```
{
    "status" : 0,
    "scheme" : "challenge",
    "message" : "Please enter your response",
    "challenge" : "ABBACD",
    "token" : "REQID:20003339189"
}
```

There are four possible values for scheme:

- `otp-generated` - A One-Time-Password generated by a user device
- `otp-requested` - A One-Time-Password sent to the user

- `challenge` - A challenge/response based on a token displayed to the user
- `external` - A request to a 3rd-party prompting method, like an app-based push notification

The check-auth phase (auth-check-2fa)

This phase performs the verification step for the authentication flow initialized by `init-auth`. If the scheme is "external", the `auth-2fa-check` trigger is called to query the status of the prompt from the authentication provider. Otherwise the user is prompted for her or his One-Time-Password or challenge response, which is passed to the `auth-2fa-check` trigger via STDIN and is validated against the second factor authentication provider. The response of this trigger is represented in JSON as a status field and an option message to the user. The status values are `0` for success and non-zero for failure (authentication rejected).

If the scheme is "external", it is possible that the authentication provider might still be waiting for the user's response. Returning a status value of `2` instructs the server to neither accept or reject the authentication attempt. For example:

```
{
    "status" : 2,
    "message" : "A token was sent to your phone"
}
```

Variables for the three mandatory triggers

All three trigger's specific variables are:

`%user%` - the user's username

`%fullname%` - the user's fullname

`%email%` - the user's email address

`%host%` - the user's host's IP address

`%method%` - the authentication method from list-methods (can be set to "unknown")

`%scheme%` - the authentication scheme set by init-auth (can be set to "unknown")

`%token%` - the stashed token from the last init-auth (can be empty)

Given that the `%fullname%` and `%email%` fields are populated from fields in the user spec which are modifiable by default, if these are used, we recommend that you set `dm.user.allowselfupdate=0` to prevent users from modifying those fields.

Triggering to affect archiving

The `archive` trigger type is used in conjunction with the `+X` filetype modifier to replace the mechanism by which the Helix Core server archives files within the repository. Archive triggers are used for storing, managing, or generating content archived outside of the Helix server repository. See "Execution environment" on page 290 for platform-specific considerations.

The following table describes the fields of an archive trigger definition:

Field	Meaning
name	The name of the archive trigger.
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. 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 Helix Core server to run when a file matching ***path*** is found in the archive. Specify the command in a way that allows the Helix Core 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 Helix server 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 295 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

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.

Triggering with depots of type graph

To associate the trigger with a single repo named `//graphDepot1/repo8`, specify the path as `//graphDepot1/repo8/...` with `/...` at the end.

To associate the trigger with multiple repos, such as `//graphDepot1/repoA` and `//graphDepot2/repoB`, use asterisks (`*`) to specify `//graphDepot*/repo*/...` as the path.

For information about depots of type `graph`, see *Helix4Git Administrator Guide* and *Helix Core P4 Command Reference*.

Four variables apply:

- `%depotName%` - The depot the repo resides in
- `%repoName%` - The name of the repo
- `%repo%` - The repo, which has `.git` as a suffix, but otherwise is identical to `%repoName%`
- `%pusher%` - The user credited with the push

The following types of graph triggers are described in the order they would normally execute: "graph-push-start" below, "graph-push-reference" below, "graph-push-reference-complete" on the facing page"graph-push-complete " on the facing page

See also the "Example of checking a commit" on the facing page

graph-push-start

- Fires prior to any data being transferred as part of a `git push` operation through the connector
- Can enforce your workflow rules

graph-push-reference

- Fires for each reference that is being created or updated
- Can have logic to block the update, according to your workflow rules
- If the trigger fails on any reference, the entire push is canceled

A graph-push-reference trigger passes the original reference value in the `%oldValue%` variable, the new value in the `%newValue%` variable, and the reference name in `%reference%`.

When such a trigger is fired from a push to the Git Connector:

- the reference type is passed in the `%refType%` variable.
- the `%refFlags%` variable is populated with a list of actions that are being applied to the reference.

graph-push-reference-complete

- Fires **after** a reference has been created or updated as part of a `git push` operation through the connector
- Same variables as graph-push-reference
- Any trigger failures are ignored

graph-push-complete

- Fires when a git `push` of a specified repo has successfully completed
- You can use this trigger to signal that all the files are present and ready for a build, testing, or diagnostic tool.

Example of checking a commit

Suppose that the use case for a trigger is to enforce a business rule such as the following:

Your organization requires that all commits include a "Description" with a issue tracking number or BugIdNumber, such as P4-17870, or a Perforce Job Number, such as job097329.

The trigger code might be as follows.

```
#!/bin/bash
reference=$1
oldValue=$2
newValue=$3
refType=$4
pusher=$5
refFlags=$6
logFile='/home/perforce/triggers/helix4git/checkCommit.log'
time=`date`
echo "$time " >> $logFile
echo "Depot: $depotName" >> $logFile
echo "Repo: $repoName" >> $logFile
echo "Reference: $reference" >> $logFile
echo "oldValue: $oldValue" >> $logFile
echo "newValue: $newValue" >> $logFile
echo "refType: $refType" >> $logFile
echo "Pusher: $pusher" >> $logFile
echo "refFlags: $refFlags" >> $logFile
echo "" >> $logFile
requiredText="JIRA"
p4 graph cat-file commit $newValue >> $logFile
p4 graph cat-file commit $newValue | grep $requiredText >> $logFile
res=`p4 graph cat-file commit $newValue | sed -n
"/$requiredText/p"`
if [ -n "$res" ] ; then
echo "contains the $requiredText job number" >> $logFile
exit 0
else
echo "NOT contains the $requiredText job number" >> $logFile
exit 1
fi
```

To call the trigger:

```
checkCommit graph-push-reference //repo/rtest/...
"/home/perforce/triggers/helix4git/checkCommit.sh %reference%
%oldValue% %newValue% %refType% %pusher% %refFlags%"
```

Triggers for external file transfer

Helix Core server can be integrated with third-party WAN acceleration software to provide extremely fast transfer of archive files in a high latency network using Helix Core server multi-server architecture. This feature supports external archive transfer with two approaches:

- "Replica archive pull threads" on the facing page
- "Edge server submits" on page 349

Replica archive pull threads

To use external transfer with replica archive pull threads:

1. Set the following server configurables:

```
p4 configure set replica#pull.trigger.dir=/tmp/trigger
p4 configure set replica#lbr.replica.notransfer=1
p4 configure set lbr.autocompress=1
```

The `pull.trigger.dir` configurable specifies the location where the pull thread writes the temporary file to pass as `%archiveList%` to the pull-archive trigger.

> **Note**
> If a replica has the lbr.replication configurable set to `cache` and the user performs p4 sync or p4 print for a file that is not on the replica, the replica synchronously fetches the file directly. This is known as an "inline archive transfer". To instead force "external archive transfers" by the use of your pull-archive trigger, set lbr.replica.notransfer to `1`.
>
> To make new files of type `text` eligible for external archive transfers, set lbr.autocompress to `1`. This change only applies to text files that are created **after** you set lbr.autocompress to `1`.

2. Define a pull-archive trigger in the trigger table:

```
externalPull pull-archive pull "pull.sh %archiveList%"
```

to specify the trigger script that performs the archive transfers, where `%archiveList%` represents the name of a temporary file containing the list of files to transfer.

3. Configure replica archive pull threads with the `--trigger` option:

```
p4 configure set replica#startup.2="pull -u -i 1 --trigger --
batch=10"
```

> **Note**
> The optional `--min-size` and `--max-size` options enable you to partition archive pull threads for files of different sizes. For example,
>
> ```
> p4 configure set replica#startup.3="pull -u -i 1 --trigger --
> batch=5 --min-size=8192"
> ```
>
> specifies that the trigger ignores small files.

The size unit is bytes, but K, M, G, and T modifiers can also be used.

> **Important**
> To have small files handled by the standard archive pull threads and larger files handled by external file transfer, configure standard archive pull threads along with archive pull threads that use external transfer. For example,

```
p4 configure set replica#startup.4="pull -u -i 1 --batch=1000 --min-size=1 --max-size=8K"
```

specifies that small file transfers occur without using external software.

> **Tip**
> For high-latency configurations, a larger `--batch` value might improve archive transfer speed for large numbers of small files.

Edge server submits

To use external transfer for submits from an edge server to a commit server:

1. Set `rpl.submit.nocopy=1` to disable default submit archive copy:

   ```
   p4 configure set rpl.submit.nocopy=1
   ```

2. Define a edge-content trigger:

   ```
   edgeTransfer edge-content //... "submit.sh %changelist% %serverroot%"
   ```

3. If the edge-content trigger needs to write temporary files, set the `pull.trigger.dir` configurable for the edge server:

   ```
   p4 configure set edge#pull.trigger.dir=/tmp/edge-trigger
   ```

   and update the trigger table entry for the edge-content trigger with `%triggerdir%` to pass the configured temporary location to the trigger:

   ```
   edgeTransfer edge-content //... "submit.sh %changelist% %serverroot% %triggerdir%"
   ```

> **Tip**
> For sample triggers and additional details, see the Support Knowledgebase article, "External Archive Transfer using pull-archive and edge-content triggers".

Triggering on heartbeat (server responsiveness)

A heartbeat-related trigger can be part of a solution to monitor whether a server is responsive. One use case might be to alert an administration to evaluate whether to fail over from the master server to a standby server. (See "Failover" on page 191 and "Triggering on failed-over" on page 351.)

If the target server ...	the monitoring server can fire a trigger or extension of type ...
misses a response to the heartbeat for the first time (see the `net.heartbeat.missing` and net.heartbeat.interval configurables in *Helix Core P4 Command Reference*)	**`heartbeat-missing`**
resumes its response (see the `net.heartbeat.resumed` and net.heartbeat.missing.wait configurables in *Helix Core P4 Command Reference*)	**`heartbeat-resumed`**
misses consecutive responses that reach the maximum count (see **`the`** `net.heartbeat.missing.count` configurable in *Helix Core P4 Command Reference*)	**`heartbeat-dead`**

The special variable, `%`**`targetport`**`%`, specifies the serverport of the target server being monitored. This corresponds to the P4TARGET or the **`-t`** ***`target`*** value that the p4 heartbeat command uses.

To define any of these trigger types, use the p4 triggers command to set up the triggers form.

For example:

- **`hm heartbeat-missing heartbeat "perl hm.pl %targetport%"`**
- **`hr heartbeat-resumed heartbeat "perl hr.pl %targetport%"`**
- **`hd heartbeat-dead heartbeat "perl hd.pl %targetport%"`**

You can have zero, one, or more heartbeat-related triggers.

The following table describes the fields for a **`failed-over`** trigger definition:

Field	Meaning
`name`	The name of your trigger script, such as **`hm`**
`type`	Must be **`heartbeat-missing`**, **`heartbeat-resumed`**, or **`heartbeat-dead`**
`path`	Must be **`heartbeat`**
`command`	The trigger for Helix server to run when the trigger fires. The command is typically a call to a script. The command must be quoted. The command can take any arguments that your trigger can parse, including Helix server trigger variables.

Triggering on failed-over

A `failed-over` trigger can only be fired when a standby server becomes the new master and first starts up during a successful p4 failover command.

Special variables:

- `%standbyserverid%` expands to the serverID of the standby before failover
- `%standbyserverport%` expands to the P4PORT of the standby before failover

To define a failed-over trigger, use the p4 triggers command to configure the triggers form with this syntax:

`triggerName failed-over failed-over command`

You can have zero, one, or more failed-over triggers. Each failed-over trigger is asynchronous and independent of the others, so if one trigger fails, its failure has no effect on the others.

The following table describes the fields for a `failed-over` trigger definition:

Field	Meaning
`name`	The name of your trigger script, such as `testTrigger2`
`type`	Must be `failed-over`
`path`	Must be `failed-over`
`command`	The trigger for Helix server to run when the master restarts after a successful failover. Specify the command in a way that allows Helix server to locate and run the command. The command is typically a call to a script. The command must be quoted. The command can take any arguments that your trigger can parse, including Helix server trigger variables.

Note
See also "Triggering on heartbeat (server responsiveness)" on page 349.

Example

In the `Triggers:` field, your form might have something similar to this:

```
testTrigger failed-over failed-over "perl '%serverroot%'/test.pl
'%serverport%' '%serverid%' '%standbyserverid%'
'%standbyserverport%'"
testTrigger2 failed-over failed-over "perl '%serverroot%'/test.pl
'%serverport%' '%serverid%' '%standbyserverid%'
'%standbyserverport%'"
testTrigger3 failed-over failed-over "perl '%serverroot%'/test.pl
```

```
'%serverport%' '%serverid%' '%standbyserverid%'
'%standbyserverport%'"
```

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 data 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 292.

The ***maxError...*** variables refer to circumstances that prevented the server from completing a command, such as 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 **termReason** variables indicate whether the command exited early and why.

> **Note**
> Any unknown variables remain in the trigger invocation. 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**
%archiveList%	Filename containing files to be pulled	**pull-archive**
%argc%	Command argument count.	all except **archive**
%args%	Command argument string.	all except **archive**

Argument	Description	Available for type
`%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`
`%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

Argument	Description	Available for type
`%clientprog%`	The name of the user's client application. For example, P4V, P4Win	all
`%clientversion%`	The version of the user's client application.	all
`%command%`	Command name.	all except `archive`

`%depotName%`	The graph depot in which the repo resides.	`graph-push-start`
		`graph-push-reference`
		`graph-push-reference-complete`
		`graph-push-complete`

`%email%`	The user's email address. See "Triggering for multi-factor authentication (MFA)" on page 340.	`auth-pre-2fa`
		`auth-init-2fa`
		`auth-check-2fa`

`%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 316.	`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`

Argument	Description	Available for type

`%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`

`%fullname%`	The user's fullname. See "Triggering for multi-factor authentication (MFA)" on page 340.	`auth-pre-2fa`
		`auth-init-2fa`
		`auth-check-2fa`

`%groups%`	List of groups to which the user belongs, space-separated.	all except `archive`

`%host%`	The IP address of the host of the user. See "Triggering for multi-factor authentication (MFA)" on page 340.	`auth-pre-2fa`
		`auth-init-2fa`
		`auth-check-2fa`

`%intermediateService%`	A broker or proxy is present.	all except `archive`
`%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`

Argument	Description	Available for type
`%lastPushedChange%`	Last new changelist number. See "Additional triggers for push and fetch commands" on page 316.	`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`

Argument	Description	Available for type
`%method%`	The authentication method from list-methods (may be set to "unknown"). See "Triggering for multi-factor authentication (MFA)" on page 340.	
`%newValue%`	See "Triggering with depots of type graph" on page 345.	`graph-push-reference`
`%oldchangelist%`	If a changelist is renumbered on submit, this variable contains the old changelist number.	`change-commit` `push-commit`
`%oldValue%`	See "Triggering with depots of type graph" on page 345.	`graph-push-reference`
`%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 server. (If the command was sent directly, `%peerip%` matches `%clientip%`)	all

Argument	Description	Available for type
`%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)

`%pusher%`	The user credited with the push. See "Triggering with depots of type graph" on page 345.	`graph-push-start`
		`graph-push-reference`
		`graph-push-reference-complete`
		`graph-push-complete`

`%quote%`	A double quote character.	all
`%reference%`	See "Triggering with depots of type graph" on page 345.	`graph-push-reference`
`%refFlags%`		
`%refType%`		

`%repo%`	The repo, which has `.git` as a suffix, but otherwise is identical to `%repoName%`.	`graph-push-start`
		`graph-push-reference`
		`graph-push-reference-complete`
		`graph-push-complete`

`%repoName%`	The name of the repo. See "Triggering with depots of type graph" on page 345.	
`%rev%`	Revision of archive file	`archive`
`%scheme%`	The authentication scheme set by init-auth (can be set to "unknown"). See See "Triggering for multi-factor authentication (MFA)" on page 340.	`auth-init-2fa`

Argument	Description	Available for type
`%serverAddress%`	The IP address and port of the Helix Core 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 Helix Core server.	all
`%serverid%`	The value of the Helix Core server's `server.id`. See `p4 serverid` in the *Helix Core P4 Command Reference* for details.	all
`%serverip%`	The IP address of the server.	all
`%servername%`	The value of the Helix Core server's `P4NAME`.	all

Argument	Description	Available for type

`%serverport%`	The transport, IP address, and port of the Helix Core 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. It does not matter which of these types of connection the server uses.
	Note
	The `%serverport%` variable returns the P4PORT of the server where the trigger runs. For example:
	`changesubmit change-submit //... "bash change-submit.sh %serverport%"`
	`shelvesubmit shelve-submit //... "bash shelve-submit.sh %serverport%"`
		all

Argument	Description	Available for type
	Whether you are working through the commit server or an edge server, the change-submit trigger runs on the commit server. Therefore, the `%serverport%` variable for the change-submit trigger returns the **`P4PORT`** of the commit server. However, a shelve trigger runs on the server where you shelve a changelist. If you shelve a changelist through an edge server, the shelve-submit trigger runs on that edge server. This is true whether the shelve is being promoted or not. Therefore, in this case, the `%serverport%` variable for the shelve-submit trigger returns the **`P4PORT`** of that edge server.	
`%serverroot%`	The **`P4ROOT`** directory of the Helix Core server.	all

Argument	Description	Available for type
`%serverservices%`	A string specifying the role of the server. One of the following: - `standard` - `replica` - `broker` - `proxy` - `commit-server` - `edge-server` - `forwarding-replica` - `build-server` - `P4AUTH` - `P4CHANGE`	all except `archive`
`%serverVersion%`	Version string for the server that terminated if the command exited early. Reason for termination is given in `%termReason%`.	all except `archive`
`%specdef%`	Expanded to the spec string of the form in question.	`form`
`%standbyserverid%`	Expands to the serverID of the standby before failover	`failed-over`
`%standbyserverport%`	expands to the P4PORT of the standby before failover	`failed-over`

Argument	Description	Available for type
`%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 *Helix Core P4 Command Reference*.	**change-submit** **change-content** **change-commit** Not available for **push-*** triggers.
`%targetport%`	The serverport of the target server being monitored. Corresponds to the P4TARGET or the **-t *target*** value that the p4 heartbeat command uses.	**heartbeat-missing** **heartbeat-resumed** **heartbeat-dead**

Argument	Description	Available for type
`%terminated%`	The value of `0` indicates that the command completed. A value of `1` indicates that the command did not complete.	all except `archive`
`%termReason%`	The reason for early termination. This might be one of the following: - `'p4 monitor terminate'` - `client disconnect` - `maxScanRows` - `maxLockTime` - `maxResults` See also `%serverVersion%`.	all except `archive`
`%token%`	The stashed token from the last init-auth (can be empty). See "Triggering for multi-factor authentication (MFA)" on page 340.	`auth-init-2fa`
`%triggerdir%`	Pull.trigger.dir used for tmp files for "Triggers for external file transfer" on page 347	`edge-content`
`%triggerMeta_action%`	Command to execute when trigger is fired. Last field of trigger definition.	all except `archive`

Argument	Description	Available for type
`%triggerMeta_depotFile%`	Third field in trigger definition. Its meaning varies with the trigger type: ■ with "Change-submit triggers" on page 301, it is the path for which the trigger is expected to match. ■ with "Form-out triggers" on page 324, it might be the form type to which the trigger is expected to apply.	all except `archive`
`%triggerMeta_name%`	Trigger name: first field from trigger definition.	all except `archive`
`%triggerMeta_trigger%`	Trigger type: second field in trigger definition.	all except `archive`
`%user%`	Helix server username of the triggering user.	all

Moving a Helix Core server to a new machine

How you move an existing Helix Core 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 Helix Core server stores two types of data under the Helix server 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 Helix server-maintained binary files holding the history and present state of the versioned files. In order to move a Helix Core 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 information about the distinction between versioned files and database, as well as for an overview of backup and restore procedures in general, see "Backup and recovery" on page 175.

Also see the Support Knowledgebase article, "Moving a Helix 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 Helix Core 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 Helix Core server on any architecture. For more details, see "Creating a checkpoint" on page 177.

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

Migrating from Windows to UNIX means that both the architecture of the system *and* the CR/LF text file convention might 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 differing linefeed convention between the two platforms.

To migrate your Perforce Server between platforms with different case-sensitivity, architecture or text file formats, or to migrate your Perforce server by restoring a checkpoint, follow the instructions at the Support Knowledgebase article, Cross-Platform Perforce Server Migration.

As with all other migrations, be sure to run p4 verify after your migration.

> **Warning**
> Migrations from UNIX servers to Windows are not supported because Windows ignores case. For example, two UNIX files named **`Makefile`** and **`makefile`** would appear to be the same file on Windows.

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 147 for information on setting protections for Helix server.

If you are a licensed Helix server 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.

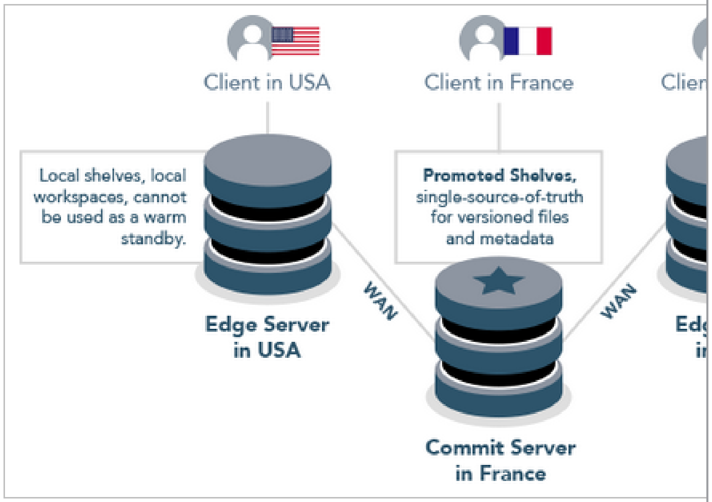
Changing the hostname of your server

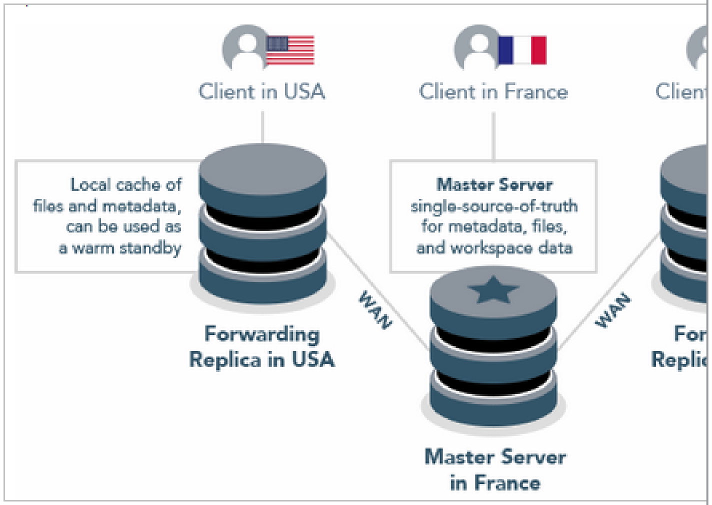
If the hostname of the new machine serving Helix server is different from that of its predecessor, your users must change their **`P4PORT`** settings. If the old machine is being retired or renamed, consider setting an alias for the new machine to match that of the old machine, so that your users won't have to change their **`P4PORT`** settings.

Deployment architecture

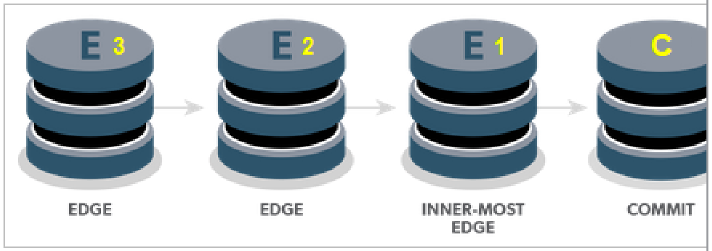
Small organizations often find a single server is adequate to meet user needs. However, as the business grows and usage expands in scale and geography, many organizations deploy a more powerful server-side infrastructure.

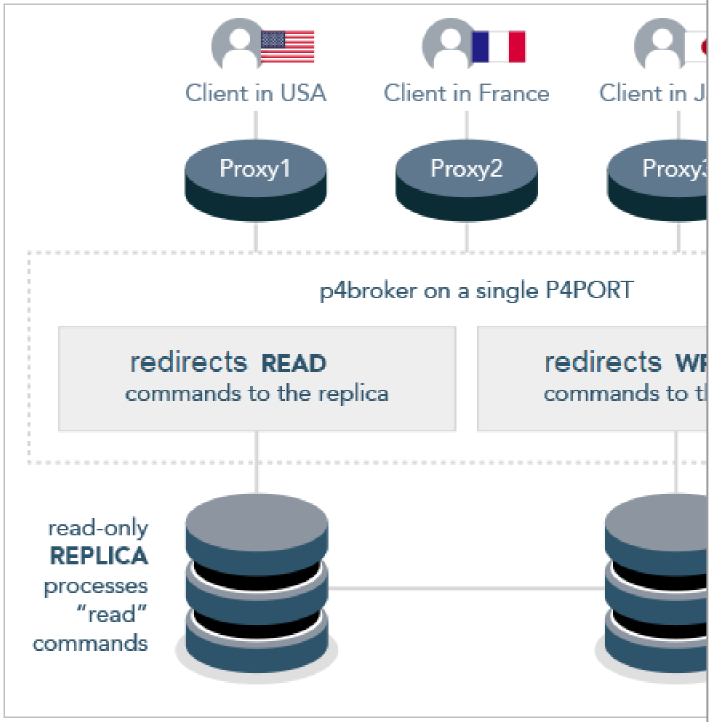
Distributed architectures

Architecture	Advantage	Disadvantage
"Commit-edge" on page 429  Client in USA Client in France Clien Local shelves, local workspaces, cannot be used as a warm standby. Promoted Shelves, single-source-of-truth for versioned files and metadata WAN WAN Edge Server in USA Commit Server in France Edg in	- best performance overall because most commands are local - an edge server that is used only for automated processing, such as builds, can be deployed without a backup/recovery solution because the edge local data is critical only during build-time.	- cannot be used as a standby - requires machine provisioning and administration, including backups of each edge (except in the case of a build edge server)

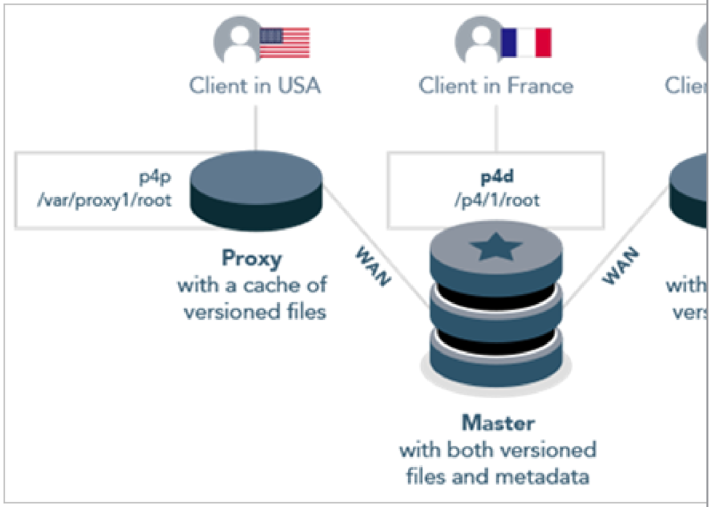
Architecture	Advantage	Disadvantage
"Forwarding replica" on page 410  Client in USA Client in France Clien Local cache of files and metadata, can be used as a warm standby Master Server single-source-of-truth for metadata, files, and workspace data WAN WAN Forwarding Replica in USA For Replic Master Server in France **Note** A master is a standard server type that doesn't support edge servers.	■ customizable filtering ■ supports "daisy chaining" to additional sites. For example, a site in the Philippines might forward to a site in Taiwan that forwards to a site in Japan that forwards to the Master in France. This alternative to directly connecting each Asian site to Europe: • reduces the metadata	■ "write" commands are slower because local metadata must be pulled from the master ■ requires machine provisioning and administration. See "Forwarding replica" on page 410. **Tip** Starting with 2018.2, we recommend a **standby** server with rpl.journalcopy.location=1 for high availability and disaster recovery.

Architecture	Advantage	Disadvantage
	replication load on the master server - reduces the amount of data traveling across the WAN from Europe to Asia For more information, see the Knowledge Base article on server-to-server arrangements, "Helix replication rules".	

Architecture	Advantage	Disadvantage
"Edge-to-edge chaining" on page 456  E 3 E 2 E 1 C EDGE EDGE INNER-MOST EDGE COMMIT	- Any number of edge servers can be chained together - If your organization is geographically dispersed, the configuration might allow all users to have an edge server nearby - Filtering	- The longer the chain, the longer it takes to complete replication - The outermost edge experiences the most latency

Architecture	Advantage	Disadvantage
"Helix Broker" on page 461  Client in USA Client in France Client in J Proxy1 Proxy2 Proxy3 p4broker on a single P4PORT redirects READ commands to the replica redirects WR commands to t read-only REPLICA processes "read" commands	- Supports the following: allow a command to **pass**, **reject** a command (for example, if the options are incorrect), **redirect** a command to another server, **filter** commands, **respond** to a command. (See "Command handler specifications" on page 474) - When the p4d process is offline for maintenance, the broker can display a message such as	- Broker layer affects network performance - Forwarding replicas and edge servers are more recent technology and are an alternative to P4Broker redirection. See the Support Knowledgebase article, "Using P4Broker to redirect read-only commands". - Command "triggers" are an alternative to some traditional broker

Architecture	Advantage	Disadvantage
	"This server is undergoing maintenance", which is more user-friendly than a TCP connect error. **Tip** Note that during the "Failover" on page 191 process, such a message is visible to the end-users without using a broker.	uses cases. See "Triggering before or after commands" on page 313.

Architecture	Advantage	Disadvantage
"Helix Proxy" on page 482  Client in USA Client in France Clie p4p /var/proxy1/root Proxy with a cache of versioned files WAN p4d /p4/1/root WAN with vers Master with both versioned files and metadata	■ easy to install, configure, and maintain ■ improves performance by caching frequently transmitted file revisions ■ reduces demand on the Perforce service and the network over which it runs ■ no need to back up the proxy cache ■ especially beneficial with larger files	■ not optimal for syncing large numbers of small files **Tip** ■ A proxy cannot be deployed in front of a forwarding replica ■ See the Support Knowledgbase article on Proxy Performance.

Services assignment

To assign a service to a server, the administrator uses the `Services:` field that appears with the p4 server command:

Tip
For additional details, see:

- "How replica types handle requests" on page 391
- the Support Knowledgebase article, "Replica Types and Use Cases"

Guidelines for setting up multi-server services

This section describes guidelines for setting up a multi-server environment.

General guidelines

This topic assumes you have read "System requirements" on page 385.

Follow these guidelines to simplify the administration and maintenance of your multi-server environments:

- Assign a **server ID** to every server
  - we recommend that the `serverID` be the same as the server name (P4NAME) if configured
  - use the p4 server command to identify each server in your network.
- Assign a **service user** name to every server by using the p4 server command
  - this simplifies the reading of logs and provides authentication and audit trails for inter-server communication
  - each service user name should be unique
  - assign service users strong passwords (see "strong passwords" at p4 password)
- Configure each server to reject operations that reduce its disk space below the limits defined by that service's `filesys.*.min` configurables, such as filesys.depot.min.
- Monitor the integrity of your replicas by using the `integrity.csv` structured server log and the `p4 journaldbchecksums` command. See "Verifying replica integrity" on page 397.

Important
Licensing for replica, edge and standby servers:

> Replica servers that are not going to be used for failover and edge servers do not require their own license if they have Helix Core server (P4D) version 2014.1 or later.
>
> Standby servers and replicas that might be required to take over from a master server do require their own license file. This can be obtained by filling out the form at https://www.perforce.com/support/duplicate-server-request.

Authenticating users

Users must have a ticket for each server they access. The best way to handle this requirement is to set up a single login to the master server, which is valid across all replica servers. This is particularly useful with failover configurations, when you would otherwise have to re-login to the new master server.

You can set up single-sign-on authentication by using two configurables:

- Set `auth.id` to a unique value for the "any" config, so it is global. This value will replace the IP address in users P4TICKETS files.
- Set `rpl.forward.login` (set to **1**) for the "any" config, so it is inherited by all replicas.

There might be a slight lag while you wait for each replica server to replicate the `db.user` record from the target server.

> **Note**
>
> - The "any" config is explained in "Viewing the values of configuration variables on all servers" in p4 configure in *Helix Core P4 Command Reference*.
> - Users will not be able to log into the server if the master/commit server is unavailable.

Connecting services

Services working together in a multi-server environment must be able to authenticate and trust one another.

- When using SSL to securely link servers, brokers, and proxies together, each link in the chain must trust the upstream link.
- It is best practice to use **ticket-based authentication** instead of password-based authentication. This means that each service user for each server in the chain must also have a valid login ticket for the upstream link in the chain. **Ticket-based authentication** is mandatory at "Server security levels" on page 130 **4** (and higher).

Managing trust between services

The user that owns the server, broker, or proxy process is typically a service user (see "User types" on page 231). As the administrator, you must create a `P4TRUST` file on behalf of the service user by using the **`p4 trust`** command. By default, a user's **`P4TRUST`** file resides in that user's home directory with **`.p4trust`** as the file name.

See "Telling Helix server applications which port to connect to" on page 44.

Managing tickets between services

When linking servers, brokers, and proxies together, each service user must be a valid service user at the upstream link, and it must be able to authenticate with a valid login ticket.

To set up service authentication:

1. On the upstream server, use `p4 user` to create a user of type **`service`**, and `p4 group` to assign it to a group that has a long or **`unlimited`** timeout.

   Use `p4 passwd` to assign the service user a strong password.
2. On the downstream server, use `p4 login` to log in to the master server as the newly-created service user, and to create a login ticket for the service user that exists on the downstream server.
3. Ensure the `P4TICKETS` configurable for the downstream server is set correctly. This enables the downstream server to correctly read the ticket file to check whether the service user is logged in to the upstream service.

Managing SSL key pairs

When configured to accept SSL connections, all server processes (**`p4d`**, **`p4p`**, **`p4broker`**), require a valid certificate and key pair on startup.

To create a key pair,

- set the directory and permissions - see `P4SSLDIR` in *Helix Core P4 Command Reference*)
- generate pairs of **`privatekey.txt`** and **`certificate.txt`** files, and make a record of the key's fingerprint:
  - on the server, use **`p4d -Gc`** to create the key/certificate pair and **`p4d -Gf`** to display its fingerprint.
  - on the broker, use **`p4broker -Gc`** to create the key/certificate pair and **`p4broker -Gf`** to display its fingerprint.
  - on the proxy, use **`p4p -Gc`** to create the key/certificate pair and **`p4p -Gf`** to display its fingerprint.

You can also supply your own private key and certificate. See "Using SSL to encrypt connections to a Helix server" on page 123.

Replication

This topic assumes you have read "Deployment architecture" on page 369.

Replication is the duplication of server data from one Helix Core server to another Helix Core server, ideally in real time. You can use replication to:

- Provide warm standby servers

  A replica server can function as an up-to-date warm standby system to be used if the master server fails. Such a replica server requires that both server metadata and versioned files are replicated.

- Reduce load and downtime on a primary server

  Long-running queries, reports, and checkpoints can be run against a replica server that only contains metadata. Situations where files are being synced or reports need access to physical archive files will mean that the replica should also have a copy of the archive files.

- Provide support for build farms

  A replica with a local (non-replicated) storage for client workspaces (and their respective have lists) is capable of running as a build farm.

- Forward write requests to a central server

  A forwarding replica holds a readable cache of both versioned files and metadata, and forwards commands that write metadata or file content towards a central server. A forwarding replica offers a blend of the functionality of the Helix Proxy with the improved performance of a replica. (See "Forwarding replica" on page 410.)

Combined with a centralized authorization server, Helix server administrators can configure the Helix Broker to redirect commands to replica servers to balance load efficiently across an arbitrary number of replica servers. See "Centralized authorization server (P4AUTH)" on page 458 and "Helix Broker" on page 461.

Note
Most replica configurations are intended for reading of data. If you require read and write access to a remote server, consider using a:

- forwarding replica - "Forwarding replica" on page 410
- multi-server Perforce service - "Commit-edge" on page 429
- the Helix Proxy - "Helix Proxy" on page 482

Tip
The following Support Knowledgebase articles contain valuable information:

- Installing a Helix Replica Server
- Checkpoints in a Distributed Helix environment

- Taking Checkpoints on Edge and Replica Servers
- Configuring Checkpoint and Rotated Journal location in Distributed Helix Environments
- Inspecting replication progress
- How to reseed a replica server
- Edge Server Meta Data Recovery
- Failing over to a replica server
- Edge Servers (differences in behavior of certain commands)

Replication basics

Commands and configurables

Replication of Helix servers depends upon certain commands, environment variables, and configurables:

Command or Feature	Typical use case
`p4 pull`	A command that can replicate both metadata and versioned files, and report diagnostic information about pending content transfers. A replica server can run multiple **`p4 pull`** commands against the same master server. To replicate both metadata and file contents, you must run two **`p4 pull`** threads simultaneously: one (and only one) **`p4 pull`** (without the **`-u`** option) thread to replicate the master server's metadata, and one (or more) **`p4 pull -u`** threads to replicate updates to the server's versioned files.
`p4 configure`	A configuration mechanism that supports multiple servers. Because **`p4 configure`** stores its data on the master server, all replica servers automatically pick up any changes you make.

Command or Feature	Typical use case
`p4 server`	A configuration mechanism that defines a server in terms of its offered services. To be effective, the **`ServerID:`** field in the **`p4 server`** form must correspond with the server's **`server.id`** file as defined by the p4 serverid command.
`p4 serverid`	A command to set or display the unique identifier for a Helix Core server. On startup, a server takes its ID from the contents of a **`server.id`** file in its root directory and examines the corresponding spec defined by the **`p4 server`** command. **Important** To avoid configuration problems, the value of `serverID` should always match the value of P4NAME if both are set. We recommend setting **`serverID`**, but support **`P4NAME`** for backward compatibility.
`p4 verify -t`	Causes the replica to schedule a transfer of the contents of any damaged or missing revisions. The command reports **`BAD!`** or **`MISSING!`** files with **`(transfer scheduled)`** at the end of the line. For the transfer to work on a replica with `lbr.replication`**`=cache`**, the replica should have one or more **`p4 pull -u`** threads configured (perhaps also using the **`--batch=`*****`N`*** flag.) **Note** You can also run the **`p4 verify -S -t`** command on a replica to request re-transfer of a **shelved** archive that is missing or bad. Re-transferring a shelved archive from the master only works **if** the shelved archive is on the master: ■ the shelf was originally created on the master, or ■ the shelf was promoted from an edge server

Command or Feature	Typical use case
Server names P4NAME	Helix servers can be identified and configured by name. When you use p4 configure on your master server, you can specify different sets of configurables for each named server. Each named server, upon startup, refers to its own set of configurables, and ignores configurables set for other servers. **Important** To avoid configuration problems, the value of `serverID` should always match the value of P4NAME if both are set. We recommend setting **`serverID`**, but support **`P4NAME`** for backward compatibility.
Service users serviceUser	A type of user intended for authentication of server-to-server communications. Service users have extremely limited access to the depot and do not consume Helix server licenses. To make logs easier to read, create one service user on your master server for each replica or proxy in your network of Helix servers .
Metadata access `db.replication`	Replica servers can be configured to automatically reject user commands that attempt to modify metadata (**`db.*`** files). In **`readonly`** mode, the Helix Core server denies any command that attempts to write to server metadata. In this mode, a command such as `p4 sync` (which updates the server's have list) is rejected, but `p4 sync` **`-p`** (which populates a client workspace *without* updating the server's have list) is accepted.
Metadata filtering `p4 server`	Replica servers can be configured to filter in (or out) data on client workspaces and file revisions. ■ To provides fine-grained control over what data is replicated, using the **`ClientDataFilter:`**, **`RevisionDataFilter:`**, and **`ArchiveDataFilter:`** fields of the `p4 server` form. • Alternatively, to explicitly filter out updates to entire database tables, use the **`-T`** ***`tableexcludelist`*** option with `p4 pull`. ■ To create a filtered checkpoint based on a **`serverId`**, use the **`p4d`** command with the **`-P`** ***`serverId`*** **`-jd`** options.

Command or Feature	Typical use case
Depot file access `lbr.replication`	Replica servers can be configured to automatically reject user commands that attempt to modify archived depot files (the “library”). ■ In `readonly` mode, the Helix Core server accepts commands that read depot files, but denies commands that write to them. In this mode, `p4 describe` can display the diffs associated with a changelist, but `p4 submit` is rejected. However, edge servers do have the capability to write some files, such as shelved files, to the depot. ■ In `shared` mode, the Helix server accepts commands that read metadata, but does not transfer new files nor remove purged files from the master. (`p4 pull -u` and `p4 verify -t`, which would otherwise transfer archive files, are disabled.) If a file is not present in the archives, commands that reference that file will fail. This mode must be used when a replica directly shares the same physical archives as the target, whether by running on the same machine or via network sharing. This mode can also be used when an external archive synchronization technique, such as `rsync`, is used for archives. ■ In `cache` mode, the Helix Core server permits commands that reference file content, but does not automatically transfer new files. Files that are purged from the target are removed from the replica when the purge operation is replicated. If a file is not present in the archives, the replica will retrieve it from the target server. ■ In `none` mode, the Helix Core server denies any command that accesses the versioned files that make up the depot. In this mode, a command such as p4 describe ***`changenum`*** is rejected because the diffs displayed with a changelist require access to the versioned files, but **`p4 describe -s`** ***`changenum`*** (which describes a changelist *without* referring to the depot files in order to generate a set of diffs) is accepted.

Command or Feature	Typical use case
Target server `P4TARGET`	As with the Helix Proxy, you can use **`P4TARGET`** to specify the master server or another replica server to which a replica server points when retrieving its data. You can set **`P4TARGET`** explicitly, or you can use `p4 configure` to set a **`P4TARGET`** for each named replica server. A replica server with **`P4TARGET`** set must have both the **`-M`** (metadata access) and **`-D`** (depot access) options. Alternatively, use the equivalent configurables: ■ `db.replication` (access to metadata) ■ `lbr.replication` (access the depot's library of versioned files)
Startup commands **`startup.1`**	Use the `startup.`*`n`* (where ***n*** is an integer) configurable to automatically spawn multiple **`p4 pull`** processes on startup.
State file `statefile`	Replica servers track the most recent journal position in a small text file that holds a byte offset. When you stop either the master server or a replica server, the most recent journal position is recorded on the replica in the state file. Upon restart, the replica reads the state file and picks up where it left off. Do not alter this file or its contents. (When the state file is written, a temporary file is used and moved into place, which should preserve the existing state file if something goes wrong when updating it. If the state file is empty or missing, the replica server will re-fetch from the start of its last used journal position.) By default, the state file is named **`state`** and it resides in the replica server's root directory. You can specify a different file name by setting the `statefile` configurable.
P4Broker	The Helix Broker can be used for load balancing, command redirection, and more. See "Helix Broker" on page 461 for details.

Warning
Replication requires uncompressed journals. Starting the master using the **`p4d -jc -z`** command breaks replication. Instead, use the **`-Z`** flag instead to prevent journals from being compressed.

System requirements

Replica servers must match the master server in the following:

- release version - see "Upgrading the server" on page 56

  > **Important**
  > **Replica servers must be at the same release level as the master server, or at a release later than the master server.**
  >
  > A given feature only works if both the master and the replica are at a release level that supports the feature.

- Unicode setting and encoding, such as UTF-8
- file system case-sensitivity
- permitted characters - for example:
  - MacOS file names cannot contain a colon (`:`)
  - Windows NTFS excludes `/ ? < > \ : * | "` and a full path is limited to 260 characters
- time zone setting
  - On Windows, the time zone setting is system-wide
  - On UNIX, the time zone setting is controlled by the `TZ` environment variable at the time the replica server is started
- A current checkpoint and versioned files from the master server are required for initial seeding of the replica server.

Additionally,

- `p4 pull` (when replicating metadata) does not read compressed journals.
  - The master server must not compress journals until the replica server has fetched all journal records from older journals.
  - Only one metadata-updating `p4 pull` thread can be active at one time.
- The replica server does not need a duplicate license file, however, if the replica serves as a failover server for the master, a duplicate license file is recommended. Fill out the form for a Helix Core Duplicate Server Request.

> **Note**
> See the details for the lbr.replication configurable.

Enabling SSL support

To encrypt the connection between a replica server and its end users, the replica must have its own valid private key and certificate pair in the directory specified by its **P4SSLDIR** environment variable. Certificate and key generation and management for replica servers works the same as it does for the (master) server. See "Enabling SSL support" on page 466. The users' Helix server applications must be configured to trust the fingerprint of the replica server.

To encrypt the connection between a replica server and its master, the replica must be configured so as to trust the fingerprint of the master server. That is, the user that runs the replica **p4d** (typically a service user) must create a **P4TRUST** file (using **p4 trust**) that recognizes the fingerprint of the *master* Helix Core server.

The **P4TRUST** variable specifies the path to the SSL trust file. You must set this environment variable in the following cases:

- for a replica that needs to connect to an SSL-enabled master server, or
- for an edge server that needs to connect to an SSL-enabled commit server.

p4 pull command

The `p4 pull` command provides the most general solution for replication. Use **p4 pull** to configure a replica server that:

- replicates versioned files (the **,v** files that contain the deltas that are produced when new versions are submitted) unidirectionally from a master server.
- replicates server metadata (the information contained in the **db.*** files) unidirectionally from a master server.
- uses the **startup.*N*** configurable to automatically spawn as many **p4 pull** processes as required.

  A common configuration for a warm standby server is one in which one (and only one) **p4 pull** process is spawned to replicate the master server’s metadata, and multiple **p4 pull -u** processes are spawned to run in parallel, and continually update the replica’s copy of the master server’s versioned files.
- The **startup.*n*** configurables are processed sequentially. Processing stops at the first gap in the numerical sequence. Any commands after a gap are ignored.

Although you can run **p4 pull** from the command line for testing and debugging purposes, it’s most useful when controlled by the **startup.*n*** configurables, and in conjunction with named servers, service users, and centrally-managed configurations.

The **--batch** option to the **p4 pull** specifies the number of files a pull thread should process in a single request. The default value of **1** is usually adequate. For high-latency configurations, a larger value might improve archive transfer speed for large numbers of small files. (Use of this option requires that both master and replica be at version 2015.2 or higher.)

Setting the **rpl.compress** configurable allows you to compress journal record data that is transmitted using **p4 pull**.

Note
If you are running a replica with monitoring enabled and you have not configured the monitor table to be disk-resident, you can run the following command to get more precise information about what pull threads are doing. (Remember to set `monitor.lsof`).

```
$ p4 monitor show -sB -la -L
```

Command output would look like this:

```
31701 B uservice-edge3 00:07:24 pull sleeping 1000 ms
    [server.locks/replica/49,d/pull(W)]
```

The possible status messages are:

For journal records	For pulling archives
`scanned NNNN records`	`sleeping`
`applied NNNN records`	`running`
`rotating journal`	

p4 pull vs. p4 replicate

Helix server also supports a more limited form of replication based on the p4 replicate command. This command does not replicate file content, but supports filtering of metadata on a per-table basis.

Shared archives between replica and master

Typical replication	Shared archives
Typically, a replica server retrieves both its metadata and file archives from the master server on the user-defined pull interval. For example `p4 pull -i 1` This configuration requires the P4TARGET server to send the archives files to the replica.	If a replica server is configured to **share** the same physical archive files as the master server: ■ the replica accesses the archives directly, so archive files are not transferred ■ the replica and master can: • run on the same machine, or • share storage over a network shared storage, where network latency affects performance.

Note
Sharing archives is supported between a master server and a replica server, or between a commit server and an edge server. Replica servers can't share archives with other replica servers.

To share archives, on the replica, set the `lbr.replication` configurable to **`shared`** either in the server spec or manually:

- only metadata is retrieved on the pull interval
- the "shared" archive files are not retrieved until requested by a client
- new files are not automatically transferred
- purged files are not removed

Shared archives can form part of a High Availability configuration.

> **Warning**
> When archive files are directly shared between a replica and master server, the replica *must* have `lbr.replication` set to **`shared`**. Otherwise, the files in the archive might be corrupted.

To configure a replica to share archive files with a master

1. Ensure that the clocks for the master and replica servers are synchronized.

   Nothing needs to be done if the master and replica servers are hosted on the same operating system.

   Synchronizing clocks is a system administration task that typically involves using a Network Time Protocol client to synchronize an operating system’s clock with a time server on the Internet, or a time server you maintain for your own network.

   See http://support.ntp.org/bin/view/Support/InstallingNTP.

2. If you have not already done so, configure the replica server as a forwarding replica.

   See "Configuring the master server for the forwarding replica" on page 410.

3. Set **`lbr.replication=shared`** either in the replica's server spec or manually using a command similar to this:

   ```
   p4 configure set fwd-1667#lbr.replication=shared
   ```

4. Restart the replica, specifying the share archive location for the replica’s root.

Result

The result of this configuration:

- archive file content is only retrieved when requested, and those requests are made against the shared archives.
- commands that would schedule the transfer of file content, such as `p4 pull -u` and `p4 verify -t` are rejected:
- if startup configurables, such as `startup.N=pull -u`, are defined, the replica server attempts to run such commands. Because the attempt to retrieve archive content is rejected, the replica’s server log will contain an error:

```
Perforce server error:
        2014/01/23 13:02:31 pid 6952 service-od@21131 background
'pull -u -i 10'
```

> **Note**
> For upgrading, see "Upgrading replica servers" on page 422.

Identifying your server

Giving your server a unique ID permits most of the server configuration data to be stored in the Helix Core server. This is an alternative to using startup options or environment variables. A unique server ID is essential for configuring replication because p4 configure settings are replicated from the master server to the replicas along with other metadata.

Configuring the following servers require the use of a server spec:

Type	Description
Commit server	central server in a distributed installation
Edge server	node in a distributed installation
Build server	replica that supports integration with a build server (or build farm)
Standby server	read-only replica that uses p4 journalcopy
Forwarding standby	forwarding replica that uses p4 journalcopy

The p4 serverid `<serverid>` command can be used to create a small text file named `server.id` in the P4ROOT directory of the server. The server executable, `p4d`, can also create this `server.id` file:

`p4d -r $P4ROOT -xD <serverid>`

> **Tip**
> - To see the server id, use `p4d -xD` or the p4 serverid command
> - If the response is `"Server does not yet have a server ID"`, set the server ID with `p4d -xD myServer`
> - To change an existing server ID, delete the `server.id` file, then set the server ID

You can use the p4 server command to:

- define a specification for each of the servers known to your installation
- create a unique server ID that can be passed to the p4 serverid command, and to define the services offered by any server that, upon startup, reads that server ID from a `server.id` file

For example, you can set your master server id to `myMaster` and the replica id to `myReplica`:

```
p4 -p svrA.company.com:11111 serverid myMaster
Server myMaster saved.
```

```
p4 -p svrB.company.com:22222 serverid myReplica
Server myReplica saved.
```

You can use p4 configure on the master to control settings on both the master and the replica because configuration settings are part of the replicated metadata of a Helix server server.

For example, if you issue the following commands on the master server:

```
$ p4 -p svrA.company.com:11111 configure set myMaster#monitor=2
$ p4 -p svrA.company.com:11111 configure set myReplica#monitor=1
```

the two servers have different monitoring levels after the configuration data has been replicated. Therefore, if you run p4 monitor `show` against the master server, you see both active and idle processes because the monitor configurable is set to `2` for the master server. In contrast, if you run p4 monitor `show` against the replica, you see only active processes because `1` is the monitor setting.

A master and each replica is likely to have its own journal and checkpoint files. To ensure their prefixes are unique, use the journalPrefix configurable for each named server:

```
$ p4 -p svrA.company.com:11111 configure set
myMaster#journalPrefix=/p4/ckps/myMaster
For server 'myMaster', configuration variable 'journalPrefix' set
to '/p4/ckps/myMaster'
```

```
$ p4 -p svrA.company.com:11111 configure set
myReplica#journalPrefix=/p4/ckps/myReplica
For server 'myReplica', configuration variable 'journalPrefix' set
to '/p4/ckps/myReplica'
```

Service users

There are three "User types" on page 231: `standard` users, `operator` users, and `service` users.

- A `standard` user is a traditional user of Helix server
- an `operator` user is intended for human or automated system administrators
- a `service` user is for server-to-server authentication as part of the replication process. Service users are:
  - useful for remote depots in single-server environments
  - required for multi-server and distributed environments
  - do not consume Helix server licenses

Create a `service` user for each master, replica, or proxy server that you control. This makes it easier to interpret your server logs. Having `service` users improves security, by requiring that your edge servers and other replicas have valid login tickets before they can communicate with the master or commit server.

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 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
```

Important
Service users *must* have a ticket created with the `p4 login` for replication to work.

Permissions for service users

On the master 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. For example:

```
super group unlimited_timeout * //..."
```

grants the super permission to the group named `unlimited_timeout`.

How replica types handle requests

Replica servers differ in how they respond to user commands:

Replica type	Global update commands	Read-only commands	Work-in-progress commands
Standby, replica	Reject	Local	Reject
Forwarding standby, forwarding replica	Forward	Local	Forward
Build server (for client creation and local sync)	p4 client	Local	p4 sync
Edge server	Forward	Local	Local
Standard server, commit server	Local	Local	Local

User requests fall into three categories, depending on the command and command options:

Global update	Read-only	Work-in-progress
p4 branch	p4 branches	p4 add
p4 change	p4 changes	p4 edit
p4 configure set	p4 configure show	p4 delete
p4 client	p4 client -o	p4 diff
	p4 clients	p4 integrate
p4 counter	p4 counters	p4 reconcile
p4 depot	p4 depots	p4 resolve
	p4 dirs	p4 revert
	p4 filelog	p4 shelve
	p4 files	p4 submit
	p4 fstat	p4 sync
	p4 fixes	p4 unshelve
p4 group	p4 groups	
	p4 interchanges	
p4 job	p4 jobs	
p4 label	p4 labels	
	p4 opened	
p4 protect		
p4 server	p4 servers	
p4 stream	p4 streams	
p4 triggers		
p4 typemap		
	p4 sizes	
p4 user	p4 user -o	
	p4 users	
	p4 where	
	p4 workspaces	

Tip
For a more detailed summary of replica server types, see the Support Knowledgebase article "Replica Types and Use Cases".

Setting P4TARGET protocol

Set `P4TARGET` to the fully-qualified domain name or IP address of the master server from which a replica server is to retrieve its data.

You can:

- set **`P4TARGET`** explicitly
- use `p4 configure` to set a **`P4TARGET`** for each named replica server
- specify on the **`p4d`** command line **`-t`** ***`protocol:host:port`***
  so that **`p4d`** examines its configuration for `startup.`*`N`* commands. If no valid `p4 pull` commands are found, **`p4d`** waits for the user to manually start a **`p4 pull`** command. If you omit a target, **`p4d`** assumes the existence of an external metadata replication source, such as `p4 replicate`. For more information, see "p4 pull vs. p4 replicate" in the "p4 pull command" on page 386 topic.

Protocol	Behavior
`<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 on/connect to an IPv4 address/port. If this fails, try IPv6.
`tcp64:`	Attempt to listen on/connect to an IPv6 address/port. 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:`	Attempt to listen on/connect to an IPv4 address/port. If this fails, try IPv6. After connecting, require SSL encryption.
`ssl64:`	Attempt to listen on/connect to an IPv6 address/port. If this fails, try IPv4. After connecting, require SSL encryption.

`P4TARGET` can be the hostname or the IP address of the host. Both IPv4 and IPv6 addresses are supported. For the **`listen`** setting, you can use the **`*`** wildcard to refer to all IP addresses, but only when you are not using CIDR notation.

If you use the `*` wildcard with an IPv6 address, you must enclose the entire IPv6 address in square brackets. For example, `[2001:db8:1:2:*]` is equivalent to `[2001:db8:1:2::]/64`. Best practice is to use CIDR notation, surround IPv6 addresses with square brackets, and avoid the `*` wildcard.

Server startup commands

You can configure Helix server to automatically run commands at startup using the `p4 configure` as follows:

`p4 configure set "servername#startup.n=command"`

Where *n* represents the order in which the commands are executed: the command specified for `startup.1` runs first, then the command for `startup.2`, and so on. See startup.*N* in *Helix Core P4 Command Reference*.

Key startup commands include `p4 pull and p4 journalcopy`.

The following example specifies:

- one pull thread for metadata
- three parallel pull threads, each for a different range of file sizes, where the pull interval is 1 second for small files and 3 seconds for large files
- updating the LDAP groups every 30 seconds:

```
startup.1=pull -i 1
startup.2=pull -u -i 1  --batch=1000 --min-size=1 --max-size=2047
startup.3=pull -u -i 2  --batch=10 --min-size=2048 --max-size=4096
startup.4=pull -u -i 3  --batch=5 --min-size=4097
startup.5=ldapsync -g -i 1800
```

Additional commands you might consider are p4 cachepurge, p4 bgtask, and p4 ldapsync.

Server options to control metadata and depot access

When you start a replica that points to a master server with `P4TARGET`, set the configurables `db.replication` (access to metadata) and `lbr.replication` (access the depot's library of versioned files) to control which Helix server commands are permitted or rejected by the replica server

Replication and protections

To apply the IP address of a replica user's workstation against the protections table, prepend the string `proxy-` to the workstation's IP address.

Important
Before you prepend the string `proxy-` to the workstation's IP address, make sure that a broker or proxy is in place.

For instance, consider an organization with a remote development site with workstations on a subnet of `192.168.10.0/24`. The organization also has a central office where local development takes place; the central office exists on the `10.0.0.0/8` subnet. A Perforce service resides in the `10.0.0.0/8` subnet, and a replica resides in the `192.168.10.0/24` subnet. Users at the remote site belong to the group `remotedev`, and occasionally visit the central office. Each subnet also has a corresponding set of IPv6 addresses.

To ensure that members of the `remotedev` group use the replica while working at the remote site, but do not use the replica when visiting the local site, add the following lines to your protections table:

```
list    group    remotedev     192.168.10.0/24              -//...
list    group    remotedev     [2001:db8:16:81::]/48        -//...

write   group    remotedev     proxy-192.168.10.0/24         //...
write   group    remotedev     proxy-[2001:db8:16:81::]/48   //...

list    group    remotedev     proxy-10.0.0.0/8             -//...
list    group    remotedev     proxy-[2001:db8:1008::]/32   -//...

write   group    remotedev     10.0.0.0/8                    //...
write   group    remotedev     [2001:db8:1008::]/32          //...
```

The first line denies `list` access to all users in the `remotedev` group if they attempt to access Helix server without using the replica from their workstations in the `192.168.10.0/24` subnet. The second line denies access in identical fashion when access is attempted from the IPV6 `[2001:db8:16:81::]/48` subnet.

The third line grants `write` access to all users in the `remotedev` group if they are using the replica and are working from the `192.168.10.0/24` subnet. Users of workstations at the remote site must use the replica. (The replica itself does not have to be in this subnet, for example, it could be at `192.168.20.0`.) The fourth line grants access in identical fashion when access is attempted from the IPV6 `[2001:db8:16:81::]/48` subnet.

Similarly, the fifth and sixth lines deny `list` access to `remotedev` users when they attempt to use the replica from workstations on the central office's subnets (`10.0.0.0/8` and `[2001:db8:1008::]/32`). The seventh and eighth lines grant write access to `remotedev` users who access the Helix server directly from workstations on the central office's subnets. When visiting the local site, users from the `remotedev` group must access the Helix server directly.

When the Perforce service evaluates protections table entries, the `dm.proxy.protects` configurable is also evaluated:

- `dm.proxy.protects` defaults to `1`, which causes the `proxy-` prefix to be prepended to all client host addresses that connect via an intermediary (proxy, broker, replica, or edge server), indicating that the connection is not direct.

- Setting `dm.proxy.protects` to `0` removes the `proxy-` prefix, which allows you to write a single set of protection entries that apply both to directly-connected clients and clients that connect via an intermediary. This is more convenient but might be less secure insofar as a connection is made using an intermediary. If you use this setting, all intermediaries must be at release 2012.1 or higher.

Enabling commands that are served by the replica, such as p4 files

The example above allows only commands that are executed by the master server to run. For example, p4 edit in a forwarding-replica scenario. Commands that are served by the replica, such as p4 files, are disallowed.

If the `dm.proxy.protects` configurable is set to its default value of `1`	To run commands against the replica directly, such as p4 files, you need:
	`write user fred 1.2.3.4 //depot/...`
	To run a command that needs to be passed over "Helix Proxy" on page 482 to the master, such as p4 sync or p4 edit, you need two entries:
	`write user fred 1.2.3.4 //depot/...`
	`write user fred proxy-1.2.3.4 //depot/...`

If `dm.proxy.protects` is set to `0`	One entry is sufficient for all commands: `write user fred 1.2.3.4 //depot/...`

Verifying replica integrity

Multi-server installations are accessed through the `p4 journaldbchecksums` command, and their behavior is controlled by three configurables: `rpl.checksum.auto`, `rpl.checksum.change`, and `rpl.checksum.table`.

When you run `p4 journaldbchecksums` against a specific database table (or the set of tables associated with one of the levels predefined by the `rpl.checksum.auto` configurable), the upstream server writes a journal note containing table checksum information. Downstream replicas, upon receiving this journal note, verify these checksums and record their results in the structured log for integrity-related events.

These checks are also performed whenever the journal is rotated. In addition, triggers allow you to take action when journals are rotated. See "Triggering on journal rotation" on page 316.

Administrators who have one or more replica servers deployed should enable structured logging for integrity events, set the `rpl.checksum.*` configurables for their replica servers, and regularly monitor the logs for integrity events.

Configuration of integrity checking

Structured server logging must be enabled on every server, with at least one log recording events of type **`integrity`**. For example:

```
$ p4 configure set serverlog.file.8=integrity.csv
```

After you have enabled structured server logging, set the following configurables to the values you want:

- rpl.checksum.auto
- rpl.checksum.change
- rpl.checksum.table

These configurables:

- determine the behavior and level of checksum computations
- allow you to persistently log information about database integrity events to the integrity log on each edge/replica server

The integrity log can be parsed using the p4 logparse command for relevant events.

Best practice for most sites is a balance between performance and log size:

`p4 configure set rpl.checksum.auto=1` (or **`2`** for additional verification that is unlikely to vary between an upstream server and its replicas.)

`p4 configure set rpl.checksum.change=2` (this setting checks the integrity of every changelist, but only writes to the log if there is an error.)

`p4 configure set rpl.checksum.table=1` (this setting instructs replicas to verify table integrity on scan or unload operations, but only writes to the log if there is an error.)

Examples of verifying replica integrity

The p4 journaldbchecksums command provides a set of tools for ensuring data integrity across a multi-server installation; the command provides the ability to:

- Compute database table checksums
- Compute changelist checksums
- Compute database table block checksums
- Unload database table content in a time-consistent fashion on master/replica or commit/edge

When the **`rpl.checksum.*`** server configurables are set, they control the behavior and invocation of **`p4 journaldbchecksums`** commands when certain server events occur, such as journal rotation or the submission of a new change to the server. The **`p4 journaldbchecksums`** command can also be run manually.

Note
The examples below assume you are familiar with the Support Knowledge article on Journal notes.

Database Table Checksums

The following form of the `p4 journaldbchecksums` command:

```
p4 journaldbchecksums [-t tableincludelist | -T tableexcludelist]
[-l N]
```

causes the server to write journal notes containing table checksum information:

```
p4 journaldbchecksums -t db.rev
@nx@ 12 1487712216 @41@ 9 -933920831 0 4 0 @db.rev@ @@ @@ @@ @@
```

Edge/Replica servers automatically verify the table checksums when processing these notes, writing the results to the server log and optionally an integrity structured log if configured:

```
Table db.rev checksums match. 2017/02/21 13:23:36 version 9:
expected 0xC8557FC1, actual 0xC8557FC1
```

```
p4 logparse -m1 -F f_table=db.rev -T 'f_date f_results'
integrity.csv
... f_date 2017/02/21 13:23:36 219149298
... f_results match
```

The result of a table checksum comparison is one of the following: `match`, `DIFFER`, or `empty`. In general, the remedy to unexpected checksum differences, whether caused by failed replication or other reasons, is to restore the edge/replica server database from a new checkpoint on the commit/master server.

about DIFFER

```
Table db.have checksums DIFFER. 2017/02/21 13:08:38 version 3:
expected 0x3BB210EE, actual 0xB1BF3E83
```

```
p4 logparse -F f_results=DIFFER -T 'f_date f_table' integrity.csv
... f_date 2017/02/21 13:08:38 203821071
... f_table db.have
```

```
Table db.ldap checksums empty. 2017/02/24 11:33:54
version 0: expected 0x0, actual 0x0.
```

```
p4 logparse -F f_results=empty -T f_table integrity.csv
... f_table db.ldap
```

Possible reasons for table checksums to `DIFFER`:

- The database structure diverged as the result of software upgrades. There are certain 'on-the-fly' upgrades that are performed against a database table when data in a table is accessed and this has the potential to generate differing checksums.

- The database structure diverged as the result of replaying a checkpoint or journal. When administrators replay journal data or journal patches using `p4d -jr`, the transactions replayed into the database are not themselves journaled. This might generate differing checksums. When replaying journal data in a distributed environment, always use `p4d -s -jr` so the replayed transactions are journaled. This enables downstream edge/replica servers to replay them as part of the normal replication process. Be aware that `p4d -jr` run against a replica server only updates the replica's database files. This might generate differing checksums.
- The database structure diverged as the result journal filtering. When filtering is active in your replication process, not all journal checksums are expected to match.

Changelist Checksums

The following form of `p4 journaldbchecksums` command:

```
p4 journaldbchecksums -c change
```

causes the server to compute a checksum of an individual submitted changelist. This checksum is written as a journal note:

```
p4 journaldbchecksums -c 12073
@nx@ 15 1487961638 @41@ 12073 1 0 0 0
@46B19358420B468668781A002BA0AC15@ @@ @@ @@ @@
```

Replica servers automatically verify the checksum of the change when processing these notes and write the results to the integrity structured log:

```
p4 logparse -F f_change=12073 -T f_results integrity.csv
... f_results match
```

Server behavior depends on the setting of the rpl.checksum.change configurable.

Database Table Block Checksums

The following form of the `p4 journaldbchecksums` command:

```
p4 journaldbchecksums -s -t tablename [ -b blocksize ][-v N]
```

Causes the server to scan the specified database table. The table is scanned in blocks. The number of records in a block is specified by the `-b` flag, which defaults to 5,000. For each block, the server computes a block checksum and writes it as a journal note:

```
p4 journaldbchecksums -s -t db.have
```

```
@nx@ 17 1487964567 @41@ 3 1 313 0 0 @db.have@
@@@//Talkhouse/build/jar/Talkhouse.jar@@ @
@@@//Jam/MAIN/src/glob.c@@ @ @2BCDA450287C03DE3433AEB6278EA4AA@ @@
```

Replica servers automatically verify these blocks when processing these notes and write output to the integrity structured log if configured:

```
p4 logparse -F 'f_table=db.have' -T 'f_results f_checkSum f_
checkSum2' integrity.csv
... f_checkSum 2BCDA450287C03DE3433AEB6278EA4AA
... f_checkSum2 D41D8CD98F00B204E9800998ECF8427E
... f_results failed
```

This command can be used with large tables on a production system because the table is unlocked between each block. Inspecting the results of the block verifications reveals the location of damage that affects only part of a database table.

Database Table Unload

The following form of `p4 journaldbchecksums` command:

```
p4 journaldbchecksums -u filename -t tablename [-v N] [-z]
```

causes the server to unload the specified database table to the specified file. The command also writes a journal note describing this action:

```
p4 journaldbchecksums -u working.txt -t db.working
@nx@ 16 1487964861 @41@ 10 0 0 0 0 @db.working@ @working.txt@ @@ @@
@@
```

Replica servers automatically unload the same table to the same file when processing these notes. If only a file name is specified with `-u`, as in the example above, the unload files are created in the P4ROOT directory of both servers. Any relative path specified with `-u` is relative to `P4ROOT`. Absolute paths to the unload file can also be used. In the case of relative or absolute path, ensure that any referenced directory paths exist on both master and replica prior to running the unload.

Unloading the tables in this way allows you to compare the contents of the table in a time-consistent fashion. This command is recommended only for tables that are small. The `-z` flag specifies that the file should be compressed.

Warnings, notes, and limitations

The following warnings, notes, and limitations apply to all configurations unless otherwise noted.

- On master servers, do not reconfigure these replica settings while the replica is running:
  - `P4TARGET`
  - `dm.domain.accessupdate`
  - `dm.user.accessupdate`
- Be careful not to inadvertently write to the replica's database. This might happen by using an `-r` option without specifying the full path (and mistakingly specifying the current path), by removing db files in `P4ROOT`, and so on. For example, when using the `p4d -r . -jc` command, make sure you are not currently in the root directory of the replica or standby in which `p4 journalcopy` is writing journal files.

- Large numbers of `Perforce password (P4PASSWD) invalid or unset` errors in the replica log indicate that the service user has not been logged in or that the `P4TICKETS` file is not writable.

  In the case of a read-only directory or `P4TICKETS` file, `p4 login` appears to succeed, but `p4 login -s` generates the "invalid or unset" error. Ensure that the `P4TICKETS` file exists and is writable by the replica server.

- Client workspaces on the master and replica servers cannot overlap. Users must be certain that their `P4PORT`, `P4CLIENT`, and other settings are configured to ensure that files from the replica server are not synced to client workspaces used with the master server, and vice versa.

- Replica servers maintain a separate table of users for each replica; by default, the `p4 users` command shows only users who have used that particular replica server. (To see the master server's list of users, use `p4 users -c`).

  The advantage of having a separate user table (stored on the replica in `db.user.rp`) is that after having logged in for the first time, users can continue to use the replica without having to repeatedly contact the master server.

- All server IDs must be unique. Manually-assigned names might be easy to remember, but in very large environments, there might be more servers than is practical to administer or remember. Use the command `p4 server -g` to create a new server specification with a numeric Server ID. Such a Server ID is guaranteed to be unique.

  Whether manually-named or automatically-generated, it is the responsibility of the system administrator to ensure that the Server ID associated with a server's `p4 server` form corresponds exactly with the `server.id` file created (and/or read) by the `p4 serverid` command.

- Users of P4V and forwarding replicas are urged to upgrade to P4V 2012.1 or higher. Helix server applications older than 2012.1 that attempt to use a forwarding replica can, under certain circumstances, require the user to log in twice to obtain two tickets: one for the first read (from the forwarding replica), and a separate ticket for the first write attempt (forwarded to the master) requires a separate ticket. This confusing behavior is resolved if P4V 2012.1 or higher is used.

- Although replicas can be chained together as of Release 2013.1, (that is, a replica's `P4TARGET` can be another replica, as well as from a central server), it is the administrator's responsibility to ensure that no loops are inadvertently created in this process. Certain multi-level replication scenarios are permissible, but pointless; for example, a forwarding replica of a read-only replica offers no advantage because the read-only replica will merely reject all writes forwarded to it. Please contact Perforce Technical Support for guidance if you are considering a multi-level replica installation.

- The `rpl.compress` configurable controls whether compression is used on the master-replica connection(s). This configurable defaults to `0`. Enabling compression can provide notable performance improvements, particularly when the master and replica servers are separated by significant geographic distances.

  Enable compression with: `p4 configure set fwd-replica#rpl.compress=1`

Filtering metadata during replication or edge-to-edge chaining

As part of an HA/DR solution, you typically want to ensure that all the metadata and all the versioned files are replicated. In most other use cases, particularly build servers and/or forwarding replicas, this leads to a great deal of redundant data being transferred.

It is often advantageous to configure your replica servers to filter data on client workspaces and file revisions. For example:

- developers working on one project at a remote site do not typically need to know the state of every client workspace at other sites where other projects are being developed
- build servers don't require access to the endless stream of changes to office documents and spreadsheets associated with a typical large enterprise

> **Note**
> Also, in the case of edge-to-edge chaining, the outer edge might need only a subset of what the inner edge has.

Excluding database tables	The simplest way to filter metadata is by using the **`-T`** ***`tableexcludelist`*** option with the `p4 pull` command. If you know, for example, that a build server has no need to refer to *any* of your users' **have** lists or the state of their client workspaces, you can filter out **`db.have`** and **`db.working`** entirely with **`p4 pull -T db.have,db.working`**. Excluding entire database tables is a coarse-grained method of managing the amount of data passed between servers, requires some knowledge of which tables are most likely to be referred to during Helix server command operations, and offers no means of control over which versioned files are replicated.
Filtering by fields	You can have fine-grained control over what data is replicated by using the **`ClientDataFilter:`**, **`RevisionDataFilter:`**, and **`ArchiveDataFilter:`** fields of the `p4 server` form. These fields enable you to replicate only a subset of the server metadata and versioned files to a replica or edge. **Note** For this feature to work, the value of the **`Services:`** field in the server spec must be a value other than **`commit-server`** or **`standard`**.

Example **Filtering out client workspace data and files**

If workspaces for users in each of three sites are named with **`site[123]-ws-`*`username`***, a replica intended to act as partial backup for users at **`site1`** could be configured as follows:

```
ServerID:       site1-1668
Name:           site1-1668
Type:           server
Services:       replica
Address:        tcp:site1bak:1668
Description:
        Replicate all client workspace data, except the states of
        workspaces of users at sites 2 and 3.
        Automatically replicate .c files in anticipation of user
        requests. Do not replicate .mp4 video files, which tend
        to be large and impose high bandwidth costs.
ClientDataFilter:
        //...
        -//site2-ws-*/...
        -//site3-ws-*/...
RevisionDataFilter:
ArchiveDataFilter:
        //....c
        -//....mp4
```

When you start the replica, your **`p4 pull`** metadata thread might resemble the following:

```
$ p4 configure set "site1-1668#startup.1=pull -i 30"
```

In this configuration, only those portions of **`db.have`** that are associated with **`site1`** are replicated. All metadata concerning workspaces associated with **`site2`** and **`site3`** is ignored.

All file-related metadata is replicated. All files in the depot are replicated, except for those ending in **`.mp4`**. Files ending in **`.c`** are transferred automatically to the replica when submitted.

To further illustrate the concept, consider a build server scenario. The ongoing work of the organization (such as code, business documents, or videos) can be stored anywhere in the depot, but this build farm is dedicated to building releasable products, and has no need to have the rest of the organization’s output:

Example Replicating metadata and file contents for a subset of a depot

Releasable code is placed into **`//depot/releases/...`** and automated builds are based on these changes. Changes to other portions of the depot, as well as the states of individual workers' client workspaces, are filtered out.

```
ServerID:       builder-1669
Name:           builder-1669
Type:           server
Services:       build-server
Address:        tcp:built:1669
Description:
        Exclude all client workspace data
        Replicate only revisions in release branches
ClientDataFilter:
        -//...
RevisionDataFilter:
        //depot/releases/...
ArchiveDataFilter:
        //depot/releases/...
```

Important
If you want to exclude a subset of paths, first put inclusionary line(s), then add the exclusionary line(s) below. For example,

```
RevisionDataFilter:
    //...
    -//depot/releases/...
```

To seed the replica, you can use a command like the following to create a filtered checkpoint:

```
$ p4d -r /p4/master -P builder-1669 -jd myCheckpoint
```

The filters specified for `builder-1669` are used in creating the checkpoint. You can then continue to update the replica using the `p4 pull` command.

When you start the replica, your `p4 pull` metadata thread might resemble the following:

```
$ p4 configure set "builder-1669#startup.1=pull -i 30"
```

Therefore, this `p4 pull` thread gets metadata for replication that excludes all client workspace data (including the **have** lists) of all users.

The `p4 pull -u` thread(s) ignore all changes on the master except those that affect revisions in the `//depot/releases/...` branch, which are the only changes of interest to a build server. The only metadata that is available is that which concerns released code. All released code is automatically transferred to the build server before any requests are made, so that when the build server performs a `p4 sync`, the sync is performed locally.

Standby and forwarding-standby server

Helix Core server 2019.1 introduced replication improvements with a standby server (or a forwarding standby server), which provide advantages over using a "Read-only replica" on page 414:

- no transactions are lost if the master fails
- the p4 pull command no longer needs to acquire database locks
- all journal records are pulled before the `p4 pull` command goes to sleep

A standby server splits the pull command process into the p4 journalcopy and p4 pull -L commands.

The `p4 journalcopy` thread:

- writes journal records to a local numbered journal file. As a result, no database tables are locked
- uses a file in the server root directory, `statejcopy`, to track the last position read in the source server journal
- updates the `statejcopy` file in standby server

In replicas other than standby servers, the pull thread updates the `state` file.

The standby journal:

- is a complete copy of the master journal
- always appears as a numbered journal, such as `journal.41`, even for the current journal

Creating a Standby Server

Prerequisite: A server from which a checkpoint can be taken. A standby server must be created by using a checkpoint of its master. Conversion of a replica of any other type to a standby is not supported. In what follows, we use the example of a commit server to act as the master for a newly-created standby server.

Preparing the commit server

1. On the commit server, add a service user (if needed) and assign its `Type` to be `service`. The service user name can be any legal Perforce user name. (See "Service users" on page 390)

   ```
   p4 user -f serviceuser
   [...]
   Type: Service
   ```

2. Save the user specification and close the editor.
3. Provide the `serviceuser` with a password:

   ```
   p4 passwd serviceuser
   ```

4. Create a `service` group, add the `serviceuser` to the group, and set the `Timeout` field to `unlimited`:

```
p4 group service
[...]
Timeout: unlimited
[...]
Users:
    serviceuser
```

5. Save the group specification and close the editor to save the user.
6. Add the service group to the protections table with **`super`** capabilities:

```
p4 protect
[...]
super group service * //...
```

7. Set up the configuration for the standby server using a server spec form. For example:

```
ServerID:      commit-standby
Type:          server
Address:       {standbyserver host}:{port number}
Services:      standby
Options:       nomandatory
ReplicatingFrom:     {commit-server-ID}
Description:    Standby server for {commit-server-ID}.
DistributedConfig:
       any#auth.default.method=ldap
       any#auth.ldap.order.1=popeye
       any#auth.ldap.userautocreate=1
       any#db.monitor.shared=8192
       ...
       any#server.allowpush=2
       P4TARGET={server:port}
       P4TICKETS={/path/to/p4tickets-file}
       P4LOG={/path/to/logfile}
       db.replication=readonly
       lbr.replication=readonly
       journalPrefix={/path/to/rotated/journal/commit-standby}
```

```
        monitor=1
        rpl.journalcopy.location=1
        serviceUser={serviceuser-name}
        startup.1=journalcopy -i 1
        startup.2=pull -L -i 1
        startup.3=pull -u -i 1
```

> **Note**
>
> Under `DistributedConfig:`
>
> - the values in **bold** and {*italics*} represent values specific to your server
> - entries starting `any#` are configurables that have already been set on the commit server to apply to all servers. These entries cannot be changed in this form.
> - The `Options:` field must be set to `nomandatory` until the standby is running and up to date. This field can then be changed to `mandatory`. See the "Failover" on page 191 topic.

8. Take a checkpoint of the commit server. You will it use to create the ***standby*** server that can become the commit during a "Failover" on page 191 process.

Preparing the standby server

1. Run the following command to restore metadata from the specified checkpoint:

   ```
   p4d r /standby/p4root -jr commit.ckp.12345
   ```

2. Set the server ID for the newly-seeded standby server:

   ```
   p4d -r /standby/p4root -xD commit-standby
   ```

3. Log the service user into the commit server from the machine the standby server is running on:

   ```
   p4 -E P4TICKETS={/path/to/p4tickets-file} -u {serviceuser-name}
   -p {commit-server:port} login
   ```

4. Start the commit standby replica.

5. Confirm the server status with the command:

   ```
   p4 -p {standbyserver:port} -Ztag info
   ```

6. Confirm that `serverServices` is set to standby and that `replica` is showing the server and port number of the commit server.

   ```
   p4 -ztag info
   [...]
   ... ServerID commit-standby
   [...]
   ... replica commit-server:1666
   ```

   > **Important**
   > If you want the standby server to be capable of becoming the new master during "Failover" on page 191, fill out a Duplicate Server Request form at https://www.perforce.com/support/duplicate-server-request. Having the license ready reduces the risk of delay during a failover process.

Monitoring a Standby Server

1. Use p4 servers -J to check the replication status of all standby replicas. For example:

   ```
   p4 servers -J
   commit-standby '2019/09/20 12:49:16' standby 1234/8779
   1234/8779 wadL/1 1
   ```

   If you want verbose output, use the `-Ztag` option:

   ```
   p4 -Ztag servers -J
   ... ServerID commit-standby
   ... Updated 2019/09/20 12:49:16
   ... ServerType standby
   ```

```
... ServerOptions nomandatory
... PersistedJournal 1234
... PersistedSequence 8779
... AppliedJournal 1234
... AppliedSequence 8779
... JAFlags wadL/1
... IsAlive 1
```

2. Use p4 journalcopy -l to determine the current copy position as detected by the replica:

   ```
   p4 journalcopy -l
   Current replica persisted journal state is: Journal 1234,
   Sequence 8779.
   ```

3. To check the status of the replica metadata replication, use the p4 pull -lj command:

   ```
   p4 pull -lj
   Current replica journal state is: Journal 1234, Sequence 8779.
   Current master journal state is: Journal 1234, Sequence 8779.
   The statefile was last modified at: 2019/09/20 12:49:16.
   The replica server time is currently: 2019/09/20 13:20:11 -0700
   PDT
   ```

Forwarding replica

A forwarding replica offers a blend of the functionality of the Helix Proxy with the improved performance of a replica.

If you are auditing server activity, each of your forwarding replica servers must have its own `P4AUDIT` log configured.

Note

- An edge server between a forwarding replica and a commit server is not supported.
- For upgrading, see "Upgrading replica servers" on page 422.

Configuring the master server for the forwarding replica

On the master server, configure the forwarding replica. We might name the forwarding replica **`fwd-1667`** because we will configure it to use port **`1667`**:

```
$ p4 server fwd-1667
```

A default server spec appears.

Spec configuration

On the master server, configure the server spec for the forwarding replica by adding some configurables and setting their values. In this example, the `ServerID` is `fwd-1667`, the replica host name is `forward`, and `forward:1667` is its `Address:`

```
ServerID:       fwd-1667
Name:           fwd-1667
Type:           server
Services:       forwarding-replica
Address:        forward:1667
DistributedConfig:
       db.replication=readonly
       lbr.replication=readonly
       lbr.autocompress=1
       startup.1=pull -i 1
       startup.2=pull -u -i 1
       startup.3=pull -u -i 1
       P4TARGET=master:1666
       serviceUser=service
       monitor=1 # optional but required if using the 'p4 monitor show'
command
       journalPrefix=/p4/journals/fw-replica # recommended
       P4TICKETS=/p4/.p4tickets # recommended
       P4LOG=/p4/logs/fw-replica.log # recommended
Description:
    Forwarding replica pointing to master:1666
```

Note

- For the optional fields, you can use your own naming conventions.
- For the `Address` field, see "Communicating port information" on page 45.
- The `DistributedConfig:` section might contain fields starting `any#`, such as

  `any#P4LOG=perforce.log`

  `any#serverlog.file.2=logs/commands.csv`

  These are options configured on the master server that, by default, apply to any server for which it is a master. To override such a configurable for the replica, add it before or after the fields containing `any#`

> For example:
>
> `any#P4LOG=perforce.log`
>
> `any#serverlog.file.2=logs/commands.csv`
>
> `P4LOG=perforce.read-only.log`
>
> `serverlog.file.2=logs/my-subdirectory/commands.csv`

Service user creation

In replicated and multi-server environments, a service user is required. See p4 user in *Helix Core P4 Command Reference*.

1. Create the service user for the replication service. For example:

   ```
   $ p4 user -f service
   ```

   The default user specification opens in your default editor. To make this user be of type `service`, add the following line:

   ```
   Type: service
   ```

2. Save the user specification and exit your default editor.
3. Use the p4 group command to create a group for your service users and set the value of the `timeout` field. To avoid service users being logged out, consider using `unlimited` as the Timeout value. See "Tickets and timeouts for service users" on page 391.
4. Set the service user group protections to `super` in your protections table. See "Service users" on page 390.
5. Set the level of security to `3` or higher on the master server. See "Server security levels" on page 130.
   For example,
   `$ p4 configure set security=4`
6. Ensure the `service` user is protected with a password:
   `$ p4 passwd service`

Next step

"Configuring the forwarding replica" below

Configuring the forwarding replica

1. Create a checkpoint of the master server.
   `p4 admin checkpoint` or `p4d -jc`

For more information, see "Backup and recovery concepts" on page 176 and "Checkpoint files" on page 176.

2. Restore from that checkpoint on the machine that the forwarding replica will run on.

   ```
   p4d -jr checkpoint_file
   ```

3. Copy the versioned files from the master server to the forwarding replica.

   On Linux, `cp -R /master/depot /replica/depot/`

   On Windows, `Xcopy /E /I C:/master/depot C:/replica/depot`

   Versioned files include both text (in RCS format, ending with `,v`) and binary files (directories of individual binary files, each directory ending with `,d`). Ensure that you copy the text files in a manner that correctly translates line endings for the forwarding replica's filesystem.

   If your depots are specified using absolute paths on the master, use the same paths on the forwarding replica. (Or use relative paths in the `Map:` field for each depot, so that versioned files are stored relative to the server's root.)

4. Start the Helix Core (p4d) server on the forwarding replica using the P4PORT value of `replica:1667` and both the `-n` and `-d` options:

   ```
   $ p4d -p forward:1667 -n -d
   ```

   > **Note**
   > The `-n` option is necessary until we set the `serverid` to correspond to a type of server that does not need a license.

5. Set the `serverid` to `fwd-1667` for the forwarding replica:

   ```
   $ p4 -p forward:1667 serverid fwd-1667
   ```

6. Confirm that the `serverid` is correctly set at the server address of `forward:1667`.

   ```
   $ p4 -p forward:1667 serverid
   ```

   The output should be:

   ```
   Server ID: fwd-1667
   ```

7. Log the service user into the master server using the location of the tickets file specified in the "Spec configuration" on page 425 section of "Configuring the master server for the forwarding replica" on page 410:

   ```
   $ p4 -p master:1666 -E P4TICKETS=/p4/.p4tickets login service
   ```

8. On the forwarding replica, stop the server:

   ```
   $ p4 -p forward:1667 admin stop
   ```

9. Restart the server on the forwarding replica:

   ```
   $ p4d -p forward:1667 -d
   ```

10. Confirm that the `p4 pull` commands specified in the `fwd-1667 startup.N` configurations are running:

```
$ p4 -p forward:1667 monitor show -a
```

The output should be similar to this:

```
18835 R service00:04:46 pull -i 1
18836 R service00:04:46 pull -u -i 1
18837 R service00:04:46 pull -u -i 1
18926 R super 00:00:00 monitor show -a
```

11. Confirm that the forwarding replica is replicating.

    ```
    $ p4 -p fwd:1667 pull -l -j
    ```

    The output should be in this format, with the replica sequence matching (or being close to) that of the master.

    ```
    Current replica journal state is:    Journal 511,    Sequence
    29233313
    Current master journal state is:    Journal 511,    Sequence
    29233313.
    The  statefile was last modified at:    2019/10/22 15:19:55.
    The replica server time is currently:    2019/10/22 15:19:59
    +0100 BST
    ```

Read-only replica

This section contains the following topics:

Master server setup for the read-only replica

On the master server, configure the read-only replica. We might name the read-only replica **`replica-1667`** because we will configure it to use port **`1667`**:

```
$ p4 server readonly-1667
```

A default server spec appears.

Spec configuration

On the master server, configure the server spec for the read-only replica by adding some configurables and setting their values. In this example, the **`ServerID`** is **`readonly-1667`**, the read-only replica host name is **`replica`**, and **`replica:1667`** is its **`Address:`**

```
ServerID:       readonly-1667
Name:           readonly-1667
```

```
Type:          server
Services:      replica
Address:       replica:1667
DistributedConfig:
       db.replication=readonly
       lbr.replication=readonly
       lbr.autocompress=1
       startup.1=pull -i 1
       startup.2=pull -u -i 1
       startup.3=pull -u -i 1
       P4TARGET=master:1666
       serviceUser=service
       monitor=1 # optional but required if using the 'p4 monitor show'
command
       journalPrefix=/p4/journals/read-only-replica # recommended
       P4TICKETS=/p4/.p4tickets # recommended
       P4LOG=/p4/logs/read-only-replica.log # recommended
Description:
       Read-only replica pointing to master:1666
```

Note

- For the optional fields, you can use your own naming conventions.
- For the **`Address`** field, see "Communicating port information" on page 45.
- The **`DistributedConfig:`** section might contain fields starting **`any#`**, such as

  `any#P4LOG=perforce.log`

  `any#serverlog.file.2=logs/commands.csv`

  These are options configured on the master server that, by default, apply to any server for which it is a master. To override such a configurable for the replica, add it before or after the fields containing **`any#`**

  For example:

  `any#P4LOG=perforce.log`

  `any#serverlog.file.2=logs/commands.csv`

  `P4LOG=perforce.read-only.log`

  `serverlog.file.2=logs/my-subdirectory/commands.csv`

Service user creation

In replicated and multi-server environments, a service user is required. See p4 user in *Helix Core P4 Command Reference*.

1. Create the service user for the replication service. For example:

   ```
   $ p4 user -f service
   ```

   The default user specification opens in your default editor. To make this user be of type `service`, add the following line:

   ```
   Type: service
   ```

2. Save the user specification and exit your default editor.
3. Use the p4 group command to create a group for your service users and set the value of the `timeout` field. To avoid service users being logged out, consider using `unlimited` as the Timeout value. See Tickets and timeouts for service users.
4. Set the service group protections to `super` in your protections table. See "Service users" on page 390.
5. Set the level of security to `3` or higher on the master server. See "Server security levels" on page 130.
   For example,
   `$ p4 configure set security=4`
6. Ensure the `service` user is protected with a password:
   `$ p4 passwd service`

Next step

"Creating the read-only replica" below

Creating the read-only replica

1. Create a checkpoint of the master server.
   `p4 admin checkpoint` or `p4d -jc`

   For more information, see "Backup and recovery concepts" on page 176 and "Checkpoint files" on page 176.

2. Restore from that checkpoint on the machine that the read-only replica will run on:
   `p4d -jr` ***`checkpoint_file`***
3. Copy the versioned files from the master server to the read-only replica. For example,

   On Linux, `scp -r master:/p4/depot/ replica:/p4/depot/`

   On Windows, ensure that the `C:` drive is shared appropriately on the replica, then run:

   `xcopy /E /I C:\p4\depot \\replica\C$\p4\depot`

Versioned files include both text (in RCS format, ending with `,v`) and binary files (directories of individual binary files, each directory ending with `,d`). Ensure that you copy the text files in a manner that correctly translates line endings for the read-only replica's filesystem.

If your depots are specified using absolute paths on the master, use the same paths on the read-only replica. (Or use relative paths in the `Map:` field for each depot, so that versioned files are stored relative to the server’s root.)

4. Start the Helix Core (p4d) server on the replica machine using the P4PORT value of `replica:1667` and both the `-n` and `-d` options:

   ```
   $ p4d -p replica:1667 -n -d
   ```

   > **Note**
   > The `-n` option is necessary until we set the `serverid` to correspond to a type of server that does not need a license.

5. Set the `serverid` for the read-only replica:

   ```
   $ p4 -p replica:1667 serverid readonly-1667
   ```

6. Confirm that the `serverid` is correctly set.

   ```
   $ p4 -p replica:1667 serverid
   ```

   The output should be:

   ```
   Server ID: readonly-1667
   ```

7. Log the service user into the master server using the location of the tickets file specified in the "Master server setup for the read-only replica" on page 414:

   ```
   $ p4 -p master:1666 -E P4TICKETS=/p4/.p4tickets login service
   ```

8. On the read-only replica, stop the server:

   ```
   $ p4 -p replica:1667 admin stop
   ```

9. Restart the server on the read-only replica.

   ```
   $ p4d -p replica:1667 -d
   ```

10. Confirm that the `p4 pull` commands specified in the `readonly-1667 startup.N` configurations are running:

    ```
    $ p4 -p replica:1667 monitor show -a
    ```

    The output should be similar to this:

    ```
    18835 R service00:04:46 pull -i 1
    18836 R service00:04:46 pull -u -i 1
    18837 R service00:04:46 pull -u -i 1
    18926 R super 00:00:00 monitor show -a
    ```

11. Confirm that the read-only replica is replicating.

    ```
    $ p4 -p replica:1667 pull -l -j
    ```

    The output should be in this format, with the replica sequence matching (or being close to) that of the master.

    ```
    Current replica journal state is:    Journal 511,    Sequence
    29233313
    Current master journal state is:    Journal 511,    Sequence
    29233313.
    The  statefile was last modified at:    2019/10/22 15:19:55.
    The replica server time is currently:    2019/10/22 15:19:59
    +0100 BST
    ```

Next step

"Testing the replica" below

Testing the replica

Testing p4 pull

To confirm that the `p4 pull` commands (specified in `Replica1`'s `startup.n` configurations) are running, issue the following command:

```
$ p4 -u super -p replica:1667 monitor show -a
18835 R service00:04:46 pull -i 1
18836 R service00:04:46 pull -u -i 1
18837 R service00:04:46 pull -u -i 1
18926 R super 00:00:00 monitor show -a
```

If you need to stop replication for any reason, use the `p4 monitor terminate` command:

```
$ p4 -u super -p replica:1667 monitor terminate 18837 process '18837'
marked for termination
```

To restart replication, either restart the Helix server process, or manually restart the replication command:

```
$ p4 -u super -p replica:1667 pull -u -i 1
```

If the `p4 pull` and/or `p4 pull -u` processes are terminated, read-only commands will continue to work for replica users as long as the replica server’s `p4d` is running.

Testing file replication

Create a new file under your workspace view:

```
$ echo "hello world" > myfile
```

Mark the file for add:

```
$ p4 -p master:1666 add myfile
```

And submit the file:

```
$ p4 -p master:1666 submit -d "testing replication"
```

Wait a few seconds for the pull commands on the replica to run, then check the replica for the replicated file:

```
$ p4 -p replica:1667 print //depot/myfile
//depot/myfile#1 - add change 1 (text)
hello world
```

If a file transfer is interrupted for any reason, and a versioned file is not present when requested by a user, the replica server silently retrieves the file from the master.

> **Note**
> Replica servers in `-M readonly -D readonly` mode will retrieve versioned files from master servers even if started without a `p4 pull -u` command to replicate versioned files to the replica. Such servers act as "on-demand" replicas, as do servers running in `-M readonly -D ondemand` mode or with their `lbr.replication` configurable set to `ondemand`.
>
> *Administrators*: be aware that creating an on-demand replica of this sort can still affect server performance or resource consumption, for example, if a user enters a command such as `p4 print //...`, which reads every file in the depot.

Verifying the replica

When you copied the versioned files from the master server to the replica server, you relied on the operating system to transfer the files. To determine whether data was corrupted in the process, run `p4 verify` on the replica server:

```
$ p4 verify //...
```

Any errors that are present on the replica but not on the master indicate corruption of the data in transit or while being written to disk during the original copy operation. (Run `p4 verify` on a regular basis, because a failover server's storage is just as vulnerable to corruption as a production server.)

Next step

"Using the read-only replica" on the facing page

Using the read-only replica

You can perform all normal operations against your master server (`p4 -p master:1666 command`). To reduce the load on the master server, direct reporting (read-only) commands to the replica (`p4 -p replica:1667 command`). Because the replica is running in `-M readonly -D readonly` mode, commands that read both metadata and depot file contents are available, and reporting commands (such as `p4 annotate`, `p4 changes`, `p4 filelog`, `p4 diff2`, `p4 jobs`, and others) work normally. However, commands that update the server's metadata or depot files are blocked.

Commands that update metadata

Some scenarios are relatively straightforward: consider a command such as `p4 sync`. A plain `p4 sync` fails, because whenever you sync your workspace, the Helix Core server must update its metadata (the "have" list, which is stored in the `db.have` table). Instead, use `p4 sync -p` to populate a workspace without updating the have list:

```
$ p4 -p replica:1667 sync -p //depot/project/...@1234
```

This operation succeeds because it does not update the server's metadata.

Some commands affect metadata in more subtle ways. For example, many Helix server commands update the last-update time that is associated with a specification (for example, a user or client specification). Attempting to use such commands on replica servers produces errors unless you use the `-o` option. For example, `p4 client` (which updates the `Update:` and `Access:` fields of the client specification) fails:

```
$ p4 -p replica:1667 client replica_client
Replica does not support this command.
```

However, `p4 client -o` works:

```
$ p4 -p replica:1667 client -o replica_client
(client spec is output to STDOUT)
```

If a command is blocked due to an implicit attempt to write to the server's metadata, consider workarounds such as those described above. (Some commands, like `p4 submit`, always fail, because they attempt to write to the replica server's depot files; these commands are blocked by the `-D readonly` option.)

Using the Helix Broker to redirect commands

You can use the Helix Broker with a replica server to redirect read-only commands to replica servers. This approach enables all your users to connect to the same `protocol:host:port` setting (the broker). In this configuration, the broker is configured to transparently redirect key commands to whichever Helix Core server is appropriate to the task at hand.

For an example of such a configuration, see the Knowledge Base article, "Using P4Broker to redirect read-only commands".

See also the chapter on "Helix Broker" on page 461.

Read-only replica as warm standby

Although it is possible for a read-only replica to be used as warm standby server, since the 2019.1 Helix Server release we recommend the use of standby servers for ease of management and failover. See "Standby and forwarding-standby server" on page 406.

However, if you want to use the pre-2019.1 technology, continue with this topic.

Pre-2019.1 information

To support warm standby servers, a replica server requires an up-to-date copy of both the master server's metadata and its versioned files.

> **Tip**
> To help the standby server stay as current as possible with the master server, consider setting the rpl.journalcopy.location configurable. The value of `1` could keep the standby server's journalcopy more current with the master server's journal by writing the journalcopy to a faster device than the device in the `journalPrefix` configurable defined for the standby server.

> **Note**
> Replication is asynchronous, and a replicated server is not recommended as the sole means of backup or disaster recovery. **We recommend that you maintain a separate set of database checkpoints and depot backups.**
>
> In addition, see the "Failover" on page 191 topic.
>
> Disaster recovery and failover strategies can be complex and site-specific, in which case Perforce Consultants are available to assist organizations in planning and deployment.

The following extended example configures a replica as a warm standby server for an existing Helix Core server with some data in it. For this example, assume that:

- Your master server is named **`Master`** and is running on a host called **`master`**, using port **`1666`**, and its server root directory is **`/p4/master`**.
- Your replica server will be named **`Replica1`** and will be configured to run on a host machine named **`replica`**, using port **`1667`**, and its root directory will be **`/p4/replica`**.
- The service user name is **`service`**.

> **Note**
> You cannot define **`P4NAME`** using the **`p4 configure`** command because a server must know its own name to use values set by **`p4 configure`**.
>
> You cannot define **`P4ROOT`** using the **`p4 configure`** command because it is important to avoid the risk of specifying an incorrect server root.

> **Important**
> To avoid configuration problems, the value of `serverID` should always match the value of P4NAME if both are set. We recommend setting **`serverID`**, but support **`P4NAME`** for backward compatibility.

Upgrading replica servers

Upgrade replicas before upgrading the master

We recommend that you first upgrade any server that replicates from a master server. If replicas are chained together, start at the replica that is furthest downstream from the master, and work upstream towards the master server. Keep downstream replicas stopped until the server immediately upstream is upgraded. Minimize the time between the upgrades.

> **Note**
> There has been a significant change in release 2013.3 that affects how metadata is stored in **`db.*`** files; despite this change, the database schema and the format of the checkpoint and the journal files between 2013.2 and 2013.3, remains unchanged.
>
> Consequently, in this one case (of upgrades between 2013.2 and 2013.3), it is sufficient to stop the replica until the master is upgraded, but the replica (and any replicas downstream of it) must be upgraded to *at least 2013.2* before a 2013.3 master is restarted.

When upgrading between 2013.2 (or lower) and 2013.3 (or higher):

- before shutting down the replica and commencing the upgrade, wait for all archive transfers to end
- before restarting the replica, you must manually delete the **`rdb.lbr`** file in the replica server’s root

Steps to upgrade a replica server in a p4 pull environment

> **Note**
> If you are changing the hostname or IP of the master server, additional steps are required.

The following process is the same for all replica types, whether you are working with a read-only replica, a forwarding replica, a build server, or commit-edge.

On the replica

1. Stop the replica server with p4 admin stop.
2. Take a checkpoint of the replica server: `p4d -r /usr/replica/root -J journal -jd checkpoint`
3. Replace the replica server's **p4d** executable.
4. Upgrade the replica database: `p4d -r /usr/replica/root -J journal -xu`
5. For each of your replica servers, repeat steps 1 - 4.

On the master

1. Stop the master server:

   `p4 admin stop`
2. Take a checkpoint of the master server and back up its versioned files:

   `p4d -r /usr/master/root -J journal -jc` ***`prefix`***

   where ***`prefix`*** is the journal prefix that the production environment uses.

   > **Note**
   > The only way to recover from failures that might occur during the upgrade process is to restore from this checkpoint.
3. Replace the master server's `p4d` executable.
4. Upgrade the master database:

   `p4d -r /usr/master/root -J journal -xu`
5. Start the upgraded master server:

   `p4d -p 1666 -r /usr/master/root -J journal -d`

On each of the upgraded replica servers

Start the server:

`p4d -p 6661 -r /usr/replica/root -J journal -d`

Build server

You can offload the workload of the automated build processes onto a separate machine. This machine is a build server, and if you use multiple build machines, you might call them a build farm.

Using one or more build servers ensures that the resources of your main Helix server (or servers) are available to your users for their normal day-to-day tasks.

Continuous integration and other similar development processes can impose a significant workload on your Helix server infrastructure. Automated build processes frequently access the Helix server to monitor recent changes and retrieve updated source files. Their client workspace definitions and associated have lists also occupy storage and memory on the server.

If your automation load exceeds the capacity of a single build server, you can configure any number of additional build servers. The term "build farm" typically implies more than one build server.

> **Note**
> Build farm servers were implemented in Helix server release 2012.1. With the implementation of edge servers in 2013.2, we now recommend that you use an edge server instead of a build server. Edge servers provide the functionality of build servers and yet offload more work from the main server. This improves performance adds the flexibility of being able to run write commands as part of the build process. See "Commit-edge" on page 429.)

A Helix Core server intended for use as a build farm must:

- Permit the creation and configuration of client workspaces
- Permit those workspaces to be synced

One issue with implementing a build server rather than a read-only replica is that under Helix server, both of those operations involve writes to metadata:

- to use a client workspace in a build environment, the workspace must contain some information specific to the build environment, such as the client workspace root.
- for a build tool to efficiently sync a client workspace, a build server must be able to keep a record of which files have already been synced.

To address these issues, build servers host their own local copies of certain metadata. In addition to the Helix server commands supported in a read-only replica environment, build servers support the `p4 client` and `p4 sync` commands when applied to workspaces that are bound to that replica.

If you are auditing server activity, each of your build servers must have its own `P4AUDIT` log configured.

> **Note**
> For upgrading, see "Upgrading replica servers" on page 422.

Configuring the master server for the build server

On the master server, configure the build server. We might name the build server **`build-1667`** because we will configure it to use port **`1667`**:

```
$ p4 server build-1667
```

A default server spec appears.

Spec configuration

On the master server, configure the server spec for the build server by adding some configurables and setting their values. In this example, the **`ServerID`** is **`build-1667`**, the build server host name is **`build`**, and its Address is **`build:1667`**:

```
ServerID:      build-1667
Name:          build-1667
Type:          server
Services:      build-server
Address:       build:1667
DistributedConfig:
       db.replication=readonly
       lbr.replication=readonly
       lbr.autocompress=1
       startup.1=pull -i 1
       startup.2=pull -u -i 1
       startup.3=pull -u -i 1
       P4TARGET=master:1666
       serviceUser=service
       monitor=1 # optional but required if using the 'p4 monitor show'
command
       journalPrefix=/p4/journals/build # recommended
       P4TICKETS=/p4/.p4tickets # recommended
       P4LOG=/p4/logs/build.log # recommended
Description:
    Build server pointing to master:1666
```

Note

- For the optional fields, you can use your own naming conventions.
- For the **`Address`** field, see "Communicating port information" on page 45.
- The **`DistributedConfig:`** section might contain fields starting **`any#`**, such as

  `any#P4LOG=perforce.log`

  `any#serverlog.file.2=logs/commands.csv`

  These are options configured on the master server that, by default, apply to any server for which it is a master. To override such a configurable for the replica, add it before or after the fields containing **`any#`**

> For example:
>
> ```
> any#P4LOG=perforce.log
> any#serverlog.file.2=logs/commands.csv
> P4LOG=perforce.read-only.log
> serverlog.file.2=logs/my-subdirectory/commands.csv
> ```

Service user creation

In replicated and multi-server environments, a service user is required. See p4 user in *Helix Core P4 Command Reference*.

1. Create the service user for the build server. For example:

   ```
   $ p4 user -f service
   ```

   The default user specification opens in your default editor. To make this user be of type `service`, add the following line:

   ```
   Type: service
   ```

2. Save the user specification and exit your default editor.
3. Use the p4 group command to create a group for your service users and set the value of the `timeout` field. To avoid service users being logged out, consider using `unlimited` as the Timeout value. See "Service users" on page 390.
4. Set the service user group protections to `super` in your protections table. See "Service users" on page 390.
5. Set the level of security to `3` or higher on the master server. See "Server security levels" on page 130.
   For example,

   ```
    $ p4 configure set security=4
   ```

6. Ensure the `service` user is protected with a password:

   ```
    $ p4 passwd service
   ```

Next step

"Configuring the build server" below

Configuring the build server

1. Create a checkpoint of the master server.
   `p4 admin checkpoint` or `p4d -jc`

For more information, see "Backup and recovery concepts" on page 176 and "Checkpoint files" on page 176.

2. Restore from that checkpoint on the machine that the build server will run on:

   ```
   p4d -jr checkpoint_file
   ```

3. Copy the versioned files from the master server to the build server.

   On Linux, `cp -R /master/depot /replica/depot/`

   On Windows, `Xcopy /E /I C:/master/depot C:/replica/depot`

   Versioned files include both text (in RCS format, ending with `,v`) and binary files (directories of individual binary files, each directory ending with `,d`). Ensure that you copy the text files in a manner that correctly translates line endings for the replica host's filesystem.

   If your depots are specified using absolute paths on the master, use the same paths on the build server. (Or use relative paths in the `Map:` field for each depot, so that versioned files are stored relative to the server's root.)

4. Start the Helix Core(p4d) server on the build server machine using the P4PORT value of `build:1667` and both the `-n` and `-d` options:

   ```
   $ p4d -p build:1667 -n -d
   ```

   > **Note**
   > The `-n` option is necessary until we set the `serverid` to correspond to a type of server that does not need a license.

5. Set the `serverid` for the build server:

   ```
   $ p4 -p build:1667 serverid build-1667
   ```

6. Confirm that the `serverid` is correctly set.

   ```
   $ p4 -p build:1667 serverid
   ```

   The output should be:

   ```
   Server ID: build-1667
   ```

7. Log the service user into the master server using the location of the tickets file you specified in the "Spec configuration" section of "Configuring the master server for the build server" on page 424:

   ```
   $ p4 -p master:1666 -E P4TICKETS=/p4/.p4tickets login service
   ```

8. On the build server, stop the server:

   ```
   $ p4 -p build:1667 admin stop
   ```

9. Restart the server on the build server:

   ```
   $ p4d -p build:1667 -d
   ```

10. Confirm that the `p4 pull` commands specified in the **`build-1667`** `startup.N` configurations are running:

    ```
    $ p4 -p build:1667 monitor show -a
    ```

    The output should be similar to this:

    ```
    18835 R service00:04:46 pull -i 1
    18836 R service00:04:46 pull -u -i 1
    18837 R service00:04:46 pull -u -i 1
    18926 R super 00:00:00 monitor show -a
    ```

11. Confirm that the build server is replicating.

    ```
    $ p4 -p build:1667 pull -l -j
    ```

    The output should be in this format, with the replica sequence matching (or being close to) that of the master.

    ```
    Current replica journal state is:    Journal 511,    Sequence
    29233313
    Current master journal state is:    Journal 511,    Sequence
    29233313.
    The  statefile was last modified at:    2019/10/22 15:19:55.
    The replica server time is currently:    2019/10/22 15:19:59
    +0100 BST
    ```

Next step

"Binding workspaces to the build server" below

Binding workspaces to the build server

At this point, there should be two servers in operation:

- a master server named **`master`**, with a server ID of **`master-1666`**
- a **`build server`** named **`build-1667`**, with a server ID of **`build-1667`**

1. Bind client workspaces to the build server.

   Because this server is configured to offer the **`build-server`** service, it maintains its own local copy of the list of client workspaces (**`db.domain`** and **`db.view.rp`**) and their respective have lists (**`db.have.rp`**).

   On the build server, create a client workspace with p4 client:

   ```
   $ p4 -c build0001 -p build:1667 client build0001
   ```

When creating a new workspace on the build server, you must ensure that your current client workspace has a **`ServerID`** that matches the **`ServerID`** required by **`build:1667`**. Because workspace **`build0001`** does not yet exist, you must manually specify **`build0001`** as the current client workspace with the **`-c`** ***`clientname`*** option and simultaneously supply **`build0001`** as the argument to the **`p4 client`** command. See the Support Knowledgebase article on "Build Farm Client Management".

When the `p4 client` form appears, set the **`ServerID:`** field to **`build-1667`**. If the **`ServerID`** is not set manually, it will be set automatically when the form is saved.

2. Sync the bound workspace.

   Because the client workspace **`build0001`** is bound to **`build-1667`**, users on the master server are unaffected. However, users on the build server are able to edit its specification and sync it:

   ```
   $ export P4PORT=build:1667
   $ export P4CLIENT=build0001
   $ p4 sync
   ```

   The build server’s have list is updated, but does not propagate back to the master.

In a real-world scenario:

- your organization’s build engineers would re-configure your site’s build system to use the new server by resetting their `P4PORT` to point directly at the build server. Even in an environment in which continuous integration and automated build tools create a client workspace (and sync it) for every change submitted to the master server, performance on the master would be unaffected.
- performance on the master is likely to improve for all users because of the reduction of read and write operations on the master server’s database.

> **Tip**
> If there are database tables that you know your build server does not require, consider using the **`-T`** filter option to p4 pull. Also consider specifying the **`ArchiveDataFilter:`**, **`RevisionDataFilter:`** and **`ClientDataFilter:`** fields of the build server’s **`p4 server`** spec form.

Commit-edge

We recommend you consider commit-edge because it can provide excellent performance in many scenarios.

This topic assumes you have read "Deployment architecture" on page 369.

Note
You cannot issue the **`p4 unsubmit`** and `p4 resubmit` commands to an edge server. You can only issue these commands to a commit server.

Tip
Commit-edge architecture builds upon Helix server replication technology. Before attempting to deploy a commit-edge configuration, read "Replication" on page 379, including the section on "Connecting services" on page 377, which includes information on "Managing SSL key pairs" on page 378.

Tip
An edge server can be used instead of a build server. If the only users of an edge server are build processes, disaster recovery is possible without backing up the local edge server-specific workspace and related information. See "Migrating from existing installations" on page 443.

Important
Some Helix Core server commands behave differently when you have edge servers. See the Support Knowledgebase article, "Edge Servers".

Setting up a commit/edge configuration

This section explains how you set up a commit/edge configuration. It assumes that you have an existing server that you want to convert to a commit server and that you are familiar with Helix server management and operation. For the sake of this example, we'll assume that the existing server is in Chicago, and that we need to set up an edge server at a remote site in Tokyo.

- **Commit server**
  ```
  P4PORT=chicago.perforce.com:1666
  P4ROOT=/chicago/p4root
  ```

- **Edge server**
  `P4PORT=tokyo.perforce.com:1666`
  `P4ROOT=/tokyo/p4root`

The setup process includes the following major steps:

1. Create the service user accounts.
2. Configure the servers.
3. Create and start the servers.

You must have `super` privileges to perform these steps.

> **Tip**
> To improve performance, consider using the configurable lbr.autocompress.
>
> See also the Support Knowledgebase articles on performance.

Create service user accounts for the commit and edge server

To support secure communication between the commit server and the edge server, a user account of type service must be created. Although you can use a generic service user name for multiple edge servers, in this example we use a unique service user name for the one edge server.

1. Create a `service` user account for the commit server:

   First, issue the command:

   ```
   $ p4 user -f svc_chicago_commit
   ```

   Then, in the user spec, insert a line that sets the user `Type:` field to `service`:

   ```
   User: svc_chicago_commit
   Type: service
   ```

   Save and close the user spec.

2. Create `service` user account for the edge server:

   First, issue the command:

   ```
   $ p4 user -f svc_tokyo_edge
   ```

   Then, in the user spec, insert a line that sets the user `Type:` field to `service`:

   ```
   User: svc_tokyo_edge
   Type: service
   ```

   Save and close the user spec.

3. To prevent the service user logins from timing out, add the service users to a group with an unlimited timeout:

First, issue the command:

```
$ p4 group no_timeout
```

Then, in the `group` spec, set the `Timeout:` field to `unlimited`

Add a line under `Users:` field for `svc_chicago_commit`

followed by another line for `svc_tokyo_edge`

So that the group spec contains the following:

```
Group:  no_timeout
...
Timeout:       unlimited
...
Users:
svc_chicago_commit
svc_tokyo_edge
```

where `...` replaces lines of the spec that do not require changes.

Save and close the group spec.

4. Assign passwords to the service user accounts by providing a value at the prompts.

```
$ p4 passwd svc_chicago_commit
$ p4 passwd svc_tokyo_edge
```

5. In the protect spec, assign `super` protections to the `svc_chicago_commit` and `svc_tokyo_edge` service users.

```
$ p4 protect
super user svc_chicago_commit * //...
super user svc_tokyo_edge * //...
```

Next step

"Create commit and edge server configurations" below

Create commit and edge server configurations

> **Note**
> If your server version is prior to 2016.1, see the Knowledge Base article on "Setting up a commit/edge server environment".

The following steps are for server versions 2016.1 and later.

> **Important**
> To avoid configuration problems, the value of serverID should always match the value of P4NAME if both are set. We recommend setting **serverID**, but support **P4NAME** for backward compatibility.

1. On the commit server, create the commit server specification:

   ```
   $ p4 server -c commit-server chicago_commit
   ```

   and modify the **DistributedConfig** section to contain:

   ```
   serviceUser=svc_chicago_commit
   monitor=2
   lbr.autocompress=1
   journalPrefix=/chicago/backup/p4d_backup
   P4TICKETS=/chicago/p4root/.p4tickets
   P4LOG=/chicago/logs/chicago_commit.log
   ```

   where:

   - **serviceUser** is the name of the service user account that will be used for communication with edge servers
   - **monitor=2** enables monitoring of active commands and idle connections on this commit server with **p4 monitor show**
   - **lbr.autocompress=1** enables compressed storge for RCS file types on the commit server and is recommended in commit-edge environments for optimal archive file replication performance
   - **journalPrefix** is the prefix path used for the location and name of commit server checkpoints and rotated journals. When replicating metadata, edge servers periodically need to locate and read rotated commit server journals. The value of **journalPrefix** identifies the name and location of those journals.
   - **P4TICKETS** contains the path to the tickets file used by the commit server **serviceUser** when communicating with edge servers.
     If edge servers use SSL, configure P4TRUST for the commit server by adding:

     ```
     P4TRUST=/chicago/p4root/.p4trust
     ```

     to the **DistributedConfig** to define a trust file location used by the commit server **serviceUser** when communicating with edge servers.
   - **P4LOG** contains the path to the commit server's log file

2. On the commit server, set the **server ID** of the commit server:

   ```
   $ p4 serverid chicago_commit
   ```

3. On the commit server, create the edge server specification:

```
$ p4 server -c edge-server tokyo_edge
```

and set the ExternalAddress to the P4PORT that will be used by the edge server. (If the edge server uses ssl, the port must include the ssl prefix.)

Also, modify the `DistributedConfig` section to contain:

```
db.replication=readonly
lbr.replication=readonly
lbr.autocompress=1
rpl.compress=4
startup.1=pull -i 1
startup.2=pull -u -i 1
startup.3=pull -u -i 1
P4TARGET=chicago.perforce.com:1666
serviceUser=svc_tokyo_edge
monitor=1
journalPrefix=/tokyo/backup/p4d_backup
P4TICKETS=/tokyo/p4root/.p4tickets
P4LOG=/tokyo/logs/tokyo_edge.log
```

where

- The `db.replication=readonly` and `lbr.replication=readonly` values indicate the edge server will replicate metadata and archive data from the commit server
- `lbr.autocompress=1` enables compressed storge for RCS file types on this edge server and is recommended in commit-edge environments for optimal archive file replication performance
- `rpl.compress=4` enables compression of journal data sent by the commit server to this edge server and is recommended if the edge server is remote to the commit server
    - For edge servers that are local to the commit server, compression can be disabled by removing the `rpl.compress=4` from the `DistributedConfig` before saving
- The `startup.N` values define one metadata (`pull -i 1`) and two archive (`pull -u -i 1`) pull threads that the edge server will run on startup to facilitate metadata and archive replication
- `P4TARGET` specifies the `P4PORT` (host:port) of the commit server this edge server will replicate from
- `serviceUser` is the name of the service user account that will be used for communication with commit server
- `monitor=1` enables monitoring of active commands on this edge server with `p4 monitor show`
- `journalPrefix` is the prefix path used for the location and name of edge server checkpoints and rotated journals

- `P4TICKETS` contains the path to the tickets file used by the edge server `serviceUser` when communicating with commit server.
  If the commit server uses SSL, configure P4TRUST for the edge server by adding:

  ```
  P4TRUST=/tokyo/p4root/.p4trust
  ```

  to the `DistributedConfig` to define a trust file location used by the edge server `serviceUser` when communicating with the commit server.
- `P4LOG` contains the path to the server log file used by the edge server

Next step

"Create and start the edge servers" below

Create and start the edge servers

Now that the commit server configuration is complete, we can seed the edge server from a commit server checkpoint and complete a few more steps to create it.

1. Take a checkpoint of the commit server, and use `-K` to filter out the database content not needed by an edge server. (The `-z` flag creates a zipped checkpoint.)

   ```
   $ p4d -r /chicago/p4root -K <LIST OF TABLES> -z -jd edge.ckp
   ```

   using the appropriate string for the *<LIST OF TABLES>*

Release	<LIST OF TABLES>
2020.2 - 2021.1	"db.have,db.working,db.locks,db.resolve,db.revsh,db.workingx,db.resolvex,db.stash,db.haveg,db.workingg,db.locksg,db.resolveg,db.storagesh,db.storagesx"
2019.1 - 2020.1	"db.have,db.working,db.locks,db.resolve,db.revsh,db.workingx,db.resolvex,db.stash,db.haveg,db.workingg,db.locksg,db.resolveg,db.storagesh"
2018.2	"db.have,db.working,db.locks,db.resolve,db.revsh,db.workingx,db.resolvex,db.stash,db.haveg,db.workingg,db.locksg,db.resolveg"

Release	<LIST OF TABLES>
2015.1 - 2018.1	"db.have,db.working,db.locks,db.resolve,db.revsh,db.workingx,db.resolvex,db.stash"
2013.2 - 2014.2	"db.have,db.working,db.locks,db.resolve,db.revsh,db.workingx,db.resolvex"

2. Recover the zipped checkpoint into the edge server `P4ROOT` directory.

   ```
   $ p4d -r /tokyo/p4root -z -jr edge.ckp.gz
   ```

3. Set the server ID for the newly seeded edge server:

   ```
   $ p4d -r /tokyo/p4root -xD tokyo_edge
   ```

4. To enable the tokyo edge service user to connect to the chicago commit server, create a login ticket for the **`svc_tokyo_edge`** service user in the `P4TICKETS` file configured for the tokyo edge server. If the commit server uses SSL, trust must first be established:

   ```
   $ p4 -E P4TRUST=/tokyo/p4root/.p4trust -u svc_tokyo_edge -p
   ssl:chicago.perforce.com:1666 trust
   ```

   before creating the service user login ticket and explicitly indentifying this P4TRUST file:

   ```
   $ $ p4 -E P4TRUST=/tokyo/p4root/.p4trust -E
   P4TICKETS=/tokyo/p4root/.p4tickets -u svc_tokyo_edge -p
   ssl:chicago.perforce.com:1666 login
   ```

5. Copy the versioned files from the commit server to the edge server. Files and directories can be moved using rsync, tar, ftp, a network copy, or any other method that preserves the files as they were on the original server.

For Linux:

Run `rsync` (or the equivalent):

```
cd /chicago/p4root rsync -avz ./depot
perforce@tokyo.perforce.com:/tokyo/p4root
```

where

- `/chicago/p4root` is the commit server root
- `./depot` is one of the directories to be copied on the original server
- `perforce@tokyo.perforce.com` is the user and hostname of the new edge server
- `/tokyo/p4root` is the Helix Server root directory on the new edge server

Copy over all the versioned file directories.

For Windows:

Run `xcopy` (or the equivalent):

```
cd <Perforce original root>
cd depot
xcopy *.* S:\perforce /s /d /c /e /i /h /y
```

where

`S:\perforce` is the network drive that contains the corresponding directory on the new server.

Copy over all the versioned file directories.

> **Note**
> For Linux and Windows:
>
> - It is possible to copy most of the files before the server move, then update the versioned files later. To update the versioned files, run the same `rsync` command. The `rsync` flags used by this command will only transfer files updated since the command was last run.
> - If you do not know where the versioned files are located, run the command: p4 depots. For each depot listed, run the command: `p4 depot -o depot` and look at the `Map:` field for the depot versioned files location.

6. Start the edge server using syntax appropriate for your platform.

   For example:

```
$ p4d -r /tokyo/p4root -d
```

See the installation/upgrading instructions for "Starting the Helix server" on page 48 and "Starting and stopping the Perforce service" on page 51 in "Upgrading the server" on page 56.

7. Check the status of replication by running the following command against the edge server.

```
$ p4 pull -lj
```

8. At the commit server host machine, to enable the chicago commit service user to connect to the tokyo edge server, create a login ticket for the `svc_chicago_commit` service user in the `P4TICKETS` file configured for the chicago commit server.
   If the edge server uses SSL, trust must first be established:

```
$ p4 -E P4TRUST=/chicago/p4root/.p4trust -u svc_chicago_commit -p
ssl:tokyo.perforce.com:1666 trust
```

before creating the service user login ticket:

```
$ p4 -E P4TICKETS=/chicago/p4root/.p4tickets -u svc_chicago_
commit -p ssl:tokyo.perforce.com:1666 login
```

> **Note**
> If your severs are connected through a Secure Sockets Layer (SSL) / Transport Layer Security (TLS) cryptographic protocol, see
>
> - "Connecting services" on page 377
> - The Knowledge Base article, "SSL and TLS Protocol Versions".

Shortcuts to configuring the server

You can also configure an edge or commit server using the `-c` option to the `p4 server` command. When you specify this option, the `DistributedConfig` field of the server spec is mostly filled in for the commands that need to be run to configure the server. The workflow is as follows:

1. Open a server spec using syntax like the following

```
$ p4 server [-c edge-server|commit-server] serverId
```

For example,

```
$ p4 server -c edge-server mynewedge
```

2. Complete the `DistributedConfig` field by specifying the settings you want to configure the server. When invoked with the `-c` option, the field looks like the code shown below.

   Specified values are set appropriately for the type of server you specified in the `p4 server` command. Values marked `<unset>` must be set. Values marked `#optional` can be set if desired.

```
db.replication=readonly
lbr.replication=readonly
lbr.autocompress=1
rpl.compress=4
startup.1=pull -i 1
startup.2=pull -u -i 1
startup.3=pull -u -i 1
P4TARGET=<unset>
serviceUser=<unset>
monitor=1 # optional
journalPrefix=<unset> # optional
P4TICKETS=<unset> #optional
P4LOG=<unset> # optional
```

3. After you have saved changes, you can execute a command like the following to see the settings for the `DistributedConfig` field:

```
$ p4 server -o mynewedge
```

```
DistributedConfig:
    db.replication=readonly
    lbr.replication=readonly
    startup.1=pull -i 1
    startup.2=pull -u -i 1
    startup.3=pull -u -i 1
    P4TARGET=localhost:20161
    serviceUser=service
```

Creating a client from a template

You might want to create a client from a template when you want to create a client that is similar to an existing client (especially the view map). For example, you want to create a client that maps the mainline server code so that you can build it yourself. This might require multiple view map entries, so you want to base your client on one that already has those view maps defined.

Clients created on a commit server can be used as templates by clients created on the commit server or on any edge server.

A client bound to an edge server can be used as a template for clients on that same edge server. To use it as a template on a different edge server or on the commit server, its view map should be global (that is, copied to the commit server).

A client’s view map is made global when the client is modified and **`server.global.client.views=1`** on both the edge server to which it is bound and on the commit server. You can create a client for an edge server or commit server based on an existing client template (bound to a different edge server) using a command like the following:

```
$ p4 client -t clientBoundToOtherEdge clientBoundToMyEdge
```

The newly created client will have its **`View`** map copied from the **`View`** map of the template client, with the client name on the right-hand side entries changed from the template client name (**`clientBoundToOtherEdge`**) to the new client name (**`clientBoundToMyEdge`**).

Client workspaces and client views

Binding workspaces to the edge server

Bind client workspaces to the edge server.

Because this server is configured to offer the edge server service, it maintains its own local copy of the list of client workspaces (**`db.view`**) and their respective have lists (**`db.have`**).

On the edge server, create a client workspace with p4 client:

```
$ p4 -c edge0001 -p edge:1667 client edge0001
```

When creating a new workspace on the edge server, you must ensure that your current client workspace has a **`ServerID`** that matches that required by **`edge:1667`**. Because workspace **`edge0001`** does not yet exist, you must manually specify **`edge0001`** as the current client workspace with the **`-c`** ***`clientname`*** option and simultaneously supply **`edge0001`** as the argument to the **`p4 client`** command.

When the p4 client form appears, set the **`ServerID:`** field to **`edge-1667`** and note that if it is not set manually, it will be set automatically when the form is saved.

Setting global client views

The **`server.global.client.views`** configurable determines whether the view maps of a non-stream client on an edge server are made global when the client is modified. This configurable can be set globally or individually for each server, thus allowing client maps to be global on most edge servers while keeping them local on those edge servers that don’t need or want them to be global.

The value of **`server.global.client.views`** on an edge server determines whether it forwards view maps to a commit server.

You should make client view maps on a replica global if up-to-date information is needed by another server running a command that needs a client view map; for example, if that client is to be used as a template on another server.

- If **`server.global.client.views=1`** on an edge server, then when a client is modified on that edge server, its view map is made global.

- The default value of `0` on the edge server means that client view maps on that edge server are not made global when a client is modified.

Setting this configurable does not immediately make client view maps global; that happens only when a client is modified afterwards. Clearing this configurable does not delete the view maps of any clients, but it does prevent subsequent changes to a client's view map from being propagated to other servers. If a client with global view maps is deleted, its view maps are also deleted globally regardless of the value of `server.global.client.views`; this is to prevent orphaned view maps.

In summary, view maps of a client are made global only under these conditions:

- The client is bound to an edge server.
- The edge server has `server.global.client.views=1`.
- The client is a non-stream client.
- The client is modified.

  If you are working with an existing client, you can "modify" it by adding a few words to the description. For example, you can add a statement that this client's view maps are now global.

> **Note**
> Clients bound directly to a commit server have their view maps replicated everywhere independently of the setting of `server.global.client.views`.
>
> For complicated reasons, it is best to choose one setting for this configurable, and not change it.

Moving clients and labels

To move clients and labels between servers in a commit/edge environment, use the p4 unload and p4 reload commands.

Client Examples

The general form of the command to move a client between servers is:

```
p4 reload -c client -p P4PORT
p4 reload -c client -p serverID
```

where:

- P4PORT or `serverID` specifies the server the client will be moved from
- the server the p4 reload command is directed at specifies the server the client is moved to
- If a `serverID` is specified, the server spec for that server must contain the correct `P4PORT` value in its `Address:` field

The process is a single step with more recent versions of Helix Core server:

2014.2 and later	2014.1 and earlier
The `p4 reload` commands above implicitly run the p4 unload command against the server the client is being moved from before processing the reload.	1. Run `p4 unload -c client` against the server the client is being moved from. 2. Run `p4 reload -c -p` against the server the client is being moved to.

User `bruno` moves client `projectX_dev` from a remote edge server to his local edge server:

```
p4 -p edge_local:1666 reload -c projectX_dev -p edge_remote:1666
Client projectX_dev unloaded.
Client projectX_dev reloaded.
```

Helix Core Administrator `admin` moves client `bruno_dev` from the commit server to an edge server:

```
p4 -p edge:1666 reload -f -c bruno_dev -p commit:1666
Client bruno_dev unloaded.
Client bruno_dev reloaded.
```

where the `-f` option for `p4 reload` is required for administrators to work with clients they don't own.

Helix Core Administrator `admin` moves client `bruno_dev` from the edge server to the commit server:

```
p4 -p commit:1666 reload -f -c bruno_dev -p edge:1666
Client bruno_dev unloaded.
Client bruno_dev reloaded.
```

where:

- the `-f` option for `p4 reload` is required for administrators to work with clients they don't own
- The commit server must have a configured service user
- The service user must be logged in from the commit to the edge

  Otherwise the following error will occur:

  ```
  p4 -p commit:1666 reload -f -c bruno_dev -p edge:1666
  Client bruno_dev unloaded.
  Access for user 'remote' has not been enabled by 'p4 protect'.
  ```

Label Examples

For labels, both **`p4 unload`** and **`p4 reload`** are required.

User ***`bruno`*** moves global label ***`bruno_hotFix`*** from the commit server to an edge server.

```
p4 -p commit:1666 unload -l bruno_hotFix
Label bruno_hotFix unloaded.

p4 -p edge:1666 reload -l bruno_hotFix -p commit:1666
Label bruno_hotFix reloaded.
```

Helix Core Administrator ***`admin`*** moves global label ***`bruno_hotFix`*** from the commit server to an edge server:

```
p4 -u admin -p commit:1666 unload -f -l bruno_hotFix
Label bruno_hotFix unloaded.

p4 -u admin -p edge:1666 reload -f -l bruno_hotFix -p commit:1666
Label bruno_hotFix reloaded.
```

where the **`-f`** option for **`p4 reload`** is required for administrators to work with clients they don't own.

Migrating from existing installations

The following sections explain how you migrate to an edge-commit architecture from an existing replicated architecture.

- "Replacing existing proxies and replicas" below explains what sort of existing replicates can be profitably replaced with edge servers.
- "Deploying commit and edge servers incrementally" on the facing page describes an incremental approach to migration.
- "Hardware, sizing, and capacity" on the facing page discusses how provisioning needs shift as you migrate to the edge-commit architecture.
- "Migration scenarios" on page 445 provides instructions for different migration scenarios.

Replacing existing proxies and replicas

If you currently use a Helix Proxy, evaluate whether it should be replaced with an edge server. If a proxy is delivering acceptable performance, then it can be left in place indefinitely. You can use proxies in front of edge servers if necessary. Deploying commit and edge servers is notably more complex than deploying a master server and proxy servers. Consider your environment carefully.

Of the three types of replicas available, forwarding replicas are the best candidates to be replaced with edge servers. An edge server provides a better solution than a forwarding replica for many use cases.

Build replicas can be replaced if necessary. If your build processes need to issue write commands other than `p4 sync`, an edge server is a good option. But if your build replicas are serving adequately, then you can continue to use them indefinitely.

Read-only replicas, typically used for disaster recovery, can remain in place. You can use read-only replicas as part of a backup plan for edge servers.

Deploying commit and edge servers incrementally

You can deploy commit and edge servers incrementally. For example, an existing master server can be reconfigured to act as a commit server, and serve in hybrid mode. The commit server continues to service all existing users, workspaces, proxies, and replicas with no change in behavior. The only immediate difference is that the commit server can now support edge servers.

Once a commit server is available, you can proceed to configure one or more edge servers. Deploying a single edge server for a pilot team is a good way to become familiar with edge server behavior and configuration.

Additional edge servers can be deployed periodically, giving you time to adjust any affected processes and educate users about any changes to their workflow.

Initially, running a commit server and edge server on the same machine can help achieve a full split of operations, which can make subsequent edge server deployments easier.

Hardware, sizing, and capacity

For an initial deployment of a distributed Perforce service, where the commit server acts in a hybrid mode, the commit server uses your current master server hardware. Over time, you might see the performance load on the commit server drop as you add more edge servers. You can reevaluate commit server hardware sizing after the first year of operation.

An edge server handles a significant amount of work for users connected to that edge server. A sensible strategy is to repurpose an existing forwarding replica and monitor the performance load on that hardware. Repurposing a forwarding replica involves the following:

- Reconfiguring the forwarding replica as an edge server.
- Creating new workspaces on the edge server or transferring existing workspaces to the edge server. Existing workspaces can be transferred using `p4 unload` and `p4 reload` commands. For details, see "Migration scenarios" on the next page.

As you deploy more edge servers, you have the option to deploy fewer edge servers on more powerful hardware, or a to deploy more edge servers, each using less powerful hardware, to service a smaller number of users.

You can also take advantage of replication filtering to reduce the volume of metadata and archive content on an edge server.

> **Note**
> An edge server maintains a unique copy of local workspace metadata, which is not shared with other edge servers or with the commit server.

Filtering edge server content can reduce the demands for storage and performance capacity.

As you transition to commit-edge architecture and the commit server is only handling requests from edge servers, you may find that an edge server requires more hardware resources than the commit server.

Migration scenarios

This section provides instructions for several migration scenarios. If you do not find the material you need, request Support.

Configuring a master server as a commit server

Scenario: You have a master server. You want to convert your master to a commit server, allowing it to work with edge servers as well as to continue to support clients.

1. Choose a ServerID for your master server, if it does not have one already, and use `p4 serverid` to save it.
2. Define a server spec for your master server or edit the existing one if it already has one, and set `Services: commit-server`.

Converting a forwarding replica to an edge server

Scenario: You currently have a master server and a forwarding replica. You want to convert your master server to a commit server and convert your forwarding replica to an edge server.

Depending on how your current master server and forwarding replica are set up, you may not have to do all of these steps.

1. Have all the users of the forwarding replica either submit, shelve, or revert all of their current work, and have them delete their current workspaces.
2. Stop your forwarding replica.
3. Choose a ServerID for your master server, if it does not have one already, and use `p4 serverid` to save it.
4. Define a server spec for your master server, or edit the existing one if it already has one, and set `Services: commit-server`.
5. Use `p4 server` to update the server spec for your forwarding replica, and set `Services: edge-server`.
6. Update the replica server with your central server data by doing one of the following:

   - Use a checkpoint:

     a. Use a dump of the database of your central server, filtering out the appropriate tables. See the `<LIST OF TABLES>` string for your server release in the "Create and start the edge servers" on page 435 topic.

     ```
     $ p4d -K <LIST OF TABLES> -jd my_filtered_checkpoint_
     file
     ```

     > **Tip**
     > If you want to produce a filtered journal dump file, look for the `-k` and `-K` options in the "Helix Core server (p4d) Reference" on page 499.

     b. Restore that checkpoint onto your replica.

     c. It is good practice, but it is not required that you remove the replica’s state file.

   - Use replication:

     a. Start your replica on a separate port (so local users don’t try to use it yet).

     b. Wait for it to pull the updates from the master.

     c. Stop the replica and remove the `<LIST OF TABLES>`.

7. Start the replica, which is now an edge server.
8. Have the users of the old forwarding replica start to use the new edge server:
   - Create their new client workspaces and sync them.

You are now up and running with your new edge server.

Converting a build server to an edge server

Scenario: You currently have a master server and a build server. You want to convert your master server to a commit server, and convert your build server to an edge server.

Build servers have locally-bound clients already, and you might want to use those clients after the conversion from a build-server to an edge server. However:

- On a build server, locally-bound clients store their *have* and *view* data in `db.have.rp` and `db.view.rp`
- On an edge server, locally-bound clients store their have and view data in `db.have` and `db.view`

Therefore the process for converting a build server to an edge server involves the following:

1. Define a ServerID and server spec for the master, setting `Services: commit-server`.
2. Edit the server spec for the build-server and change `Services: build-server` to `Services: edge-server`.

3. Shut down the build-server and do the following:

   a. Remove the tables you do not need:

      See the `<LIST OF TABLES>` string for your server release in the "Create and start the edge servers" on page 435 topic, and issue the command:

      ```
      $ rm <LIST OF TABLES>
      ```

   b. Rename the have table:

      ```
      $ mv db.have.rp  db.have
      ```

   c. Rename the view table:

      ```
      $ mv db.view.rp  db.view
      ```

4. Start the server, which is now an edge server. All of its locally-bound clients can be used.

> **Note**
> Step 3 above discards the `db.view` table, but there are multiple possibilities:
>
> 1. Retain `db.view`, Discard `db.view.rp`.
>
>   This means the edge server will discard all pre-existing build clients and need to create them in the edge server.
>
> 2. Retain `db.view.rp`, Discard `db.view`.
>
>   This means the edge server will have access to pre-existing build clients, but the other clients that were previously accessible in the build server (or build farm) become inaccessible.
>
> 3. Retain both `db.view` and `db.view.rp`.
>
>   If you want to maintain the same access of all available clients, including the build clients, request Support.

Managing commit-edge installations

Commit-edge architecture raises certain issues that you must be aware of and learn to manage.

- Each edge server maintains a unique set of workspace and work-in-progress data that must be backed up separately from the commit server. See "Backup and recovery planning" on page 454 for more information.
- Exclusive locks are global: establishing an exclusive lock requires communication with the commit server, which might incur network latency.
- Parallel submits from an edge server to a commit server use standard pull threads to transfer the files. The administrator must ensure that pull threads can be run on the commit server by doing the following:

    - Make sure that the service user used by the commit server is logged into the edge server.
    - Make sure the `ExternalAddress` field of the edge server's server spec is set to the address that will be used by the commit server's pull threads to connect to the edge server.

      If the commit and edge servers communicate on a network separate from the network used by clients to communicate with the edge server, the `ExternalAddress` field must specify the edge server ip address and port number that is used for connections from the commit server. Furthermore, the edge server must listen on the two (or more) networks.

      See the `p4 help submit` command for more information.

- Shelving changes in a distributed environment typically occurs on an edge server. Shelving can occur on a commit server only while using a client workspace bound to the commit server. Normally, changelists shelved on an edge server are not shared between edge servers.

  You can promote changelists shelved on an edge server to the commit server, making them available to other edge servers. See "Promoting shelved changelists" on the next page for details.

- Auto-creation of users is not possible on edge servers.
- You must use a command like the following to delete a client that is bound to an edge server: It is not sufficient to simply use the `-d` and `-f` options.

  ```
  $ p4 client -d -f --serverid=thatserver thatclient
  ```

  This prevents your inadvertently deleting a client from an edge server. Likewise, you must specify the server id and the changelist number when trying to delete a changelist whose client is bound to an edge server.

  ```
  $ p4 change -d -f --serverid=thatserver 6321
  ```

  > **Note**
  > An edge server that is used only for automated processing, such as builds, can be deployed without a backup/recovery solution because the edge local data is critical only during build-time.

Moving users to an edge server

As you create new edge servers, you assign some users and groups to use that edge server.

- Users need the `P4PORT` setting for the edge server.
- Users need to create a new workspace on the edge server or to transfer an existing workspace to the new edge server. Transferring existing workspaces can be automated.

If you use authentication triggers or single sign-on, install the relevant triggers on all edge servers and verify the authentication process.

Promoting shelved changelists

Changelists shelved on an edge server, which would normally be inaccessible from other edge servers, can be automatically or explicitly *promoted* to the commit server. Promoted shelved changelists are available to any edge server.

- In a shared archive configuration, where the commit server and edge servers have access to the same storage device for the archive content, shelves are automatically promoted to the commit server. See "Automatically promoting shelves" below.
- You must explicitly promote a shelf when the commit and edge servers do not share the archive. See "Explicitly promoting shelves" below.

You can view a shelf's promotion status using the **`-ztag`** output of the `p4 describe`, `p4 changes`, or `p4 change` **`-o`** commands.

See "Working with promoted shelves" on the facing page for more information on the limitations of working on promoted shelves.

Automatically promoting shelves

When the edge server and commit server are configured to access the same archive contents, shelf promotion occurs automatically, and promoting shelved files with `p4 shelve` **`-p`** is not required.

To configure the edge server and commit server to access the same archive contents, you should set `server.depot.root` to the same path for both the commit and edge server, and you should set the `lbr.replication` configurable to **`shared`** for the edge server. For example:

```
$ p4 configure set commit#server.depot.root=/p4/depot/root
$ p4 configure set edge#server.depot.root=/p4/depot/root
$ p4 configure set edge#lbr.replication=shared
```

Explicitly promoting shelves

You have two ways of explicitly promoting shelves:

- Set the `dm.shelve.promote` configurable to **`1`**

  > **Important**
  > This makes edge servers automatically promote shelved files to the commit server, which means that file content is transferred and stored both on the commit server and the edge server.
  >
  > This affects performance.
  >
  > If you are using Helix Swarm on an edge server, automatic promotion is necessary. See "Configure the Helix Server to promote all shelved changes" under "Helix Core Server configuration for Swarm" in *Helix Swarm Guide*.

- Use the **-p** option with the **p4 shelve** command.

  See the example below for more information on this option.

For example, given two edge servers, **edge1** and **edge2**:

1. Shelve and promote a changelist from **edge1**.

   ```
   edge1$ p4 shelve -p -c 89
   ```

2. The shelved changelist is now available to **edge2**.

   ```
   edge2$ p4 describe -S 89
   ```

3. Promotion is only required once.

   Subsequent **p4 shelve** commands automatically update the shelved changelist on the commit server, using server lock protection. For example, make changes on **edge1** and refresh the shelved changelist:

   ```
   edge1$ p4 shelve -r -c 89
   ```

   The updates can now be seen on **edge2**:

   ```
   edge2$ p4 describe -S 89
   ```

Promoting shelves when unloading clients

Use the **-p** option for the **p4 unload** command to promote any non-promoted shelves belonging to the specified client that is being unloaded. The shelf is promoted to the commit server where it can be accessed by other edge servers.

Working with promoted shelves

You can:

- delete the shelved files from the changelist, but you cannot unpromote a shelved changelist
- unshelve a promoted shelf into open files and branches on a server from where the shelf did not originate
- run **p4 submit -e** on a promoted shelf only on the server that owns the change
- move a promoted shelf from one edge server to another using the **p4 unshelve** command

Locking and unlocking files

You can use the **-g** flag of the **p4 lock** command to lock the files locally and globally. The **-g** option must be used with the **-c *changelist*** option. This lock is removed by the **p4 unlock -g** command or by any submit command for the specified changelist.

Use the `-x` option to the `p4 unlock` command to unlock files that have the `+l` filetype (exclusive open) but have become orphaned. This is typically only necessary in the event of an extended network outage between an edge server and the commit server.

To make `p4 lock` on an edge server take global locks on the commit server by default, set the `server.locks.global` configurable to `1`. See the section Configurables in *Helix Core P4 Command Reference*.

Triggers and commit-edge

This section explains how you manage existing "Triggers" on page 285 in a commit-edge configuration and how you use edge type triggers.

Determining the location of triggers

In a distributed Perforce service, triggers might run either on the commit server, or on the edge server, or perhaps on both.

Make sure that all relevant trigger scripts and programs are deployed appropriately. Edge servers can affect non-edge type triggers in the following ways:

- If you enforce policy with triggers, you should evaluate whether a change list or shelve trigger should execute on the commit server or on the edge server.
- Edge servers are responsible for running form triggers on workspaces and some types of labels.

> **Tip**
> Read about the sequence of triggers that run during an edge server submit in the Support Knowledgebase article, "Triggers in a Distributed Perforce Environment".

Trigger scripts can determine whether they are running on a commit or edge server using the trigger variables described in the following table. When a trigger is executed on the commit server, `%peerip%` matches `%clientip%`.

Trigger Variable	Description
`%peerip%`	The IP address of the proxy, broker, replica, or edge server.
`%clientip%`	The IP address of the machine whose user invoked the command, regardless of whether connected through a proxy, broker, replica, or edge server.
`%submitserverid%`	For a `change-submit`, `change-content`, or `change-commit` trigger in a distributed installation, the `server.id` of the edge server where the submit was run. See `p4 serverid` in the *Helix Core P4 Command Reference* for details.

Using edge triggers

In addition, edge servers support two trigger types that are specific to edge-commit architecture: `edge-submit` and `edge-content`:

Trigger Type	Description
`edge-submit`	Executes a **pre-submit** trigger on the edge server after changelist has been created, but prior to file transfer from the client to the edge server. The files are not necessarily locked at this point.
`edge-content`	Executes a **mid-submit** trigger on the edge server after file transfer from the client to the edge server, but prior to file transfer from the edge server to the commit server. At this point, the changelist is shelved.

Triggers on the edge server are executed one after another when invoked via `p4 submit -e`. For `p4 submit`, `edge-submit` triggers run immediately before the changelist is shelved, and `edge-content` triggers run immediately after the changelist is shelved.

Because `edge-submit` triggers run prior to file transfer to the edge server, these triggers cannot access file content.

The following `edge-submit` trigger is an MS-DOS batch file that rejects a changelist if the submitter has not had the change reviewed and approved. 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 "Review Approved...\n\n\t" > nul
if errorlevel 1 echo Your code has not been reviewed for changelist %1
p4 describe -s %1|findstr "Review Approved...\n\n\t" > nul
```

To use the trigger, add the following line to your triggers table:

```
sampleEdge   edge-submit //depot/qa/...   "reviewcheck.bat
%changelist%"
```

Background archive transfer for edge server submits

Users on edge servers might want to spend less time waiting for their submits to complete. Starting with 2019.1, it is possible to configure the replication environment so that:

1. The edge server sends the metadata to the commit server.
2. The user on the edge server sees the submit is complete and can resume work.
3. In the background, the commit server pulls the archive files from the edge server.

Prior to 2019.1, the user on the edge server would need to wait for both the submitted archive files and the metadata to be transferred to the commit server.

To enable background archive transfer

Prerequisites

- Ensure a service user is defined for the commit server and that this service user is logged into the **`ExternalAddress`** field of the server specification for all edge servers that will participate in background transfers.
- If any of the participating edge servers are enabled for SSL/TSL security, ensure the service user on the commit server has established trust to the **`ExternalAddress`** field for those edge servers.

Steps

1. Set the submit.allowbgtransfer configurable to **`1`** on ALL the servers .
2. Set the lbr.autocompress configurable to **`1`** on ALL the servers .

Tell your users to manually issue the `p4 submit` **`-b`** command, where the **`-b`** option causes background transfer of the archive files.

Alternatively, to enable background archive transfer with the added convenience of `p4 submit` automatically functioning as **`p4 submit -b`** so that you users do not need to use the **`-b`** option:

1. Set the submit.allowbgtransfer configurable to **`1`** on ALL the servers.
2. Set the lbr.autocompress configurable to **`1`** on ALL the servers.
3. Set the submit.autobgtransfer configurable to **`1`** on the EDGE servers.

> **Note**
> To recover a failed archive transfer, restart the transfer by using the `p4 pull` **`-u -t`** ***`target`*** command, where ***`target`*** represents the **`ExternalAddress`** of the EDGE server where the submit occurred that caused the failed transfer.
>
> For details on background file content transfers, including errors due to failed transfers, see the output of **`p4 pull -l`** against the commit server.

Backup and recovery planning

A commit server can use the same backup and high availability / disaster recovery (HA/DR) strategy as a master server. Edge servers contain unique information and should have a backup and an HA/DR plan. Whether an edge server outage is as urgent as a master server outage depends on your requirements. An edge server might have an HA/DR plan with a less ambitious Recovery Point Objective (RPO) and Recovery Time Objective (RTO) than the commit server.

If a commit server must be rebuilt from backups, each edge server must be rolled back to a backup prior to the commit server's backup.

Alternatively, if your commit server has no local users, the commit server can be rebuilt from a fully-replicated edge server. In this scenario, the edge server is a superset of the commit server.

Backing up and recovering an edge server is similar to backing up and restoring an offline replica server:

1. On the edge server, schedule a checkpoint to be taken the next time journal rotation is detected on the commit server. For example:

   ```
   $ p4 -p myedgehost:myedgeport admin checkpoint
   ```

   The `p4 pull` command performs the checkpoint at the next rotation of the journal on the commit server. A `stateCKP` file is written to the `P4ROOT` directory of the edge server, recording the scheduling of the checkpoint.
2. Rotate the journal on the commit server:

   ```
   $ p4 -p mycommithost:mycommitport admin journal
   ```

As long as the edge server's replication state file is included in the backup, the edge server can be restored and resume service. If the edge server was offline for a long period of time, it might need to catch up on the activity on the commit server.

As part of a failover plan for a commit server, make sure that the edge servers are redirected to use the new commit server.

> **Note**
> For commit servers with no local users, edge servers could take significantly longer to checkpoint than the commit server. You might want to use a different checkpoint schedule for edge servers than commit servers. If you use several edge servers for one commit server, you should stagger the edge-checkpoints so they do not all occur at once and bring the system to a stop. Journal rotations for edge servers could be scheduled at the same time as journal rotations for commit servers.

Other considerations

As you deploy edge servers, give consideration to the following areas.

- **Labels**

  In a distributed Perforce service, labels can be local to an edge server or global.

  - By default, labels are also bound to the Edge Server on which they are created.
  - The `-g` flag defaults to the value of `0`, which indicates that the label is to be defined globally on all servers in the installation. Configuring `rpl.labels.global=1` allows updating of local labels. See rpl.labels.global in the P4 Command Reference.
  - For important details, on the command line, type `p4 help distributed`.

- **Exclusive Opens**

  Exclusive opens (`+l` filetype modifier) are global: establishing an exclusive open requires communication with the commit server, which may incur network latency.

- **Integrations with third party tools**

  If you integrate third party tools, such as defect trackers, with Helix server, evaluate whether those tools should continue to connect to the master/commit server or could use an edge server instead. If the tools only access global data, then they can connect at any point. If they reference information local to an edge server, like workspace data, then they must connect to specific edge servers.

  Build processes can usefully be connected to a dedicated edge server, providing full Helix server functionality while isolating build workspace metadata. Using an edge server in this way is similar to using a build server, but with the additional flexibility of being able to run write commands as part of the build process.

- **Files with propagating attributes**

  In distributed environments, the following commands are not supported for files with propagating attributes: `p4 copy`, `p4 delete`, `p4 edit`, `p4 integrate`, `p4 reconcile`, `p4 resolve`, `p4 shelve`, `p4 submit`, and `p4 unshelve`. Integration of files with propagating attributes from an edge server is not supported; depending on the integration action, target, and source, either the `p4 integrate` or the `p4 resolve` command will fail.

  If your site makes use of this feature, direct these commands to the commit server, not the edge server. Perforce-supplied software does not presently set propagating attributes on files and is not known to be affected by this limitation.

- **Logging and auditing**

  Edge servers maintain their own set of server and audit logs. Consider using structured logs for edge servers, as they auto-rotate and clean up with journal rotations. Incorporate each edge server's logs into your overall monitoring and auditing system.

  In particular, consider the use of the `rpl.checksum.*` configurables to automatically verify database tables for consistency during journal rotation, changelist submission, and table scans and unloads. Regularly monitor the `integrity.csv` structured log for integrity events.

- **Unload depot**

The unload depot might have different contents on each edge server. Clients and labels bound to an edge server are unloaded into the unload depot on that edge server, and are not displayed by the `p4 clients -U` and `p4 labels -U` commands on other edge servers.

Be sure to include the unload depot as part of your edge server backups. The commit server does not verify that the unload depot is empty on every edge server. Therefore, to delete the unload depot from the commit server, `p4 depot -d -f` is the command.

- **Future upgrades**

  Commit and edge servers should be upgraded at the same time.

- **Time zones**

  Commit and edge servers must use the same time zone.

- **Helix Swarm**

  The initial release of Swarm can usefully be connected to a commit server acting in hybrid mode or to an edge server for the users of that edge server. Full Swarm compatibility with multiple edge servers will be handled in a follow-on Swarm release. For more detailed information about using Swarm with edge servers, please contact Perforce Technical Support support@perforce.com.

Validation

As you deploy commit and edge servers, you can focus your testing and validation efforts in the following areas.

Supported deployment configurations

- Hybrid mode: commit server also acting as a regular master server
- Read-only replicas attached to commit and edge servers
- Proxy server attached to an edge server

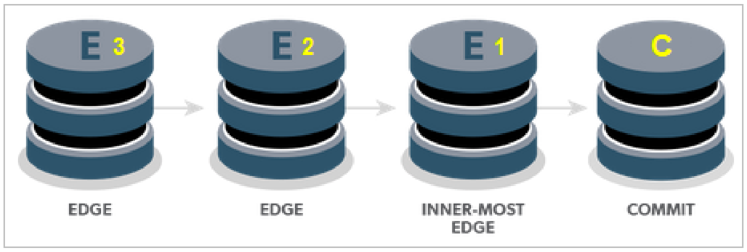
Backups

Exercise a complete backup plan on the commit and edge servers. Note that journal rotations are not permitted directly on an edge server. Journal rotations can occur on edge servers as a consequence of occurring on a master server.

Edge-to-edge chaining

If your organization is geographically dispersed, you might want all your users to have an edge server nearby.

An edge server can get files and metadata from another edge server rather than from a distant commit server.



You can have any number of such edge servers in the chain.

Configure each server so that its P4TARGET is the closest inner server, where "inner" means toward the commit server.

Log in sequence for service users

Although a regular end-user is only required to log in to the closest edge, service users must log in according to the following sequenc, where E1, E2, and E3 refer to the diagram above.

Step for E1

- Log in to the commit server.

Step for E2

- Log in to E1.

Steps for E3

1. Log in to E1.
2. Log into E2.

> **Important**
>
> - Only the inner-most edge connects directly to the commit server.
> - If you include a forwarding replica, it must connect directly with another forwarding replica or the commit server. An edge server between a forwarding replica and a commit server is not supported.

Authorization and changelist servers

As an alternative to the multi-server options at "Deployment architecture" on page 369, it is possible to centralize these functions with specialized servers that have a shared user base. Two use cases are:

- to simplify user authentication with centralized `P4AUTH`
- to guarantee unique P4CHANGE change list numbers across the organization

Centralized authorization server (P4AUTH)

If you are running multiple Helix servers, you can configure them to retrieve protections and licensing data from a *centralized authorization server*. By using a centralized server, you are freed from the necessity of ensuring that all your servers contain the same users and protections entries.

> **Note**
> When using a centralized authentication server, all outer servers must be at the same (or newer) release level as the central server.

If a user does not exist on the central authorization server, that user does not appear to exist on the outer server.

You can use any existing Helix Core server in your organization as your central authorization server. The license file for the central authorization server must be valid, as it governs the number of licensed users that are permitted to exist on outer servers. To configure a Helix Core server to use a central authorization server, set **`P4AUTH`** before starting the server, or specify it on the command line when you start the server.

If your server is making use of a centralized authorization server, the following line will appear in the output of **`p4 info`**:

```
...
Authorization Server: [protocol:]host:port
```

Where **`[protocol:]host:port`** refers to the protocol, host, and port number of the central authorization server. See "Specifying hosts" on page 470.

In the following example, an outer server is configured to use a central authorization server (named **`central`**). The outer server listens for user requests on port 1999 and relies on the central server's data for user, group, protection, review, and licensing information. It also joins the protection table from the server at **`central:1666`** to its own protections table.

For example:

```
$ p4d -a central:1666 -p 1999
```

> **Note**
> On Windows, configure the outer server with **`p4 set -S`** as follows:

```
C:\> p4 set -S "Outer Server" P4AUTH=central:1666
C:\> p4 set -S "Outer Server" P4PORT=1999
```

When you configure a central authorization server, outer servers forward the following commands to the central server for processing:

Command	Forwarded to auth server?	Note
`p4 group`	Yes	Local group data is derived from the central server.
`p4 groups`	Yes	Local group data is derived from the central server.
`p4 license`	Yes	License limits are derived from the central server. License updates are forwarded to the central server.
`p4 passwd`	Yes	Property values are derived from the central server.
`p4 property`	Yes	For example, if two Swarm instances use the same auth server, updating one instance can update the other instance.
`p4 review`	No	The default user named **`remote`** must have access to the central server. However, best practice is to create "Service users" on page 390 and not use the default user named **`remote`**. See "Restricting access to remote depots" on page 118.
`p4 reviews`	No	The default user named **`remote`** must have access to the central server. However, best practice is to create "Service users" on page 390 and not use the default user named **`remote`**. See "Restricting access to remote depots" on page 118.
`p4 user`	Yes	Local user data is derived from the central server.
`p4 users`	Yes	Local user data is derived from the central server.
`p4 protect`	No	The local server's protections table is displayed if the user is authorized (as defined by the combined protection tables) to edit it.
`p4 protects`	Yes	Protections are derived from the central server's protection table as appended to the outer server's protection table.
`p4 login`	Yes	Command is forwarded to the central server for ticket generation.

Command	Forwarded to auth server?	Note
`p4 logout`	Yes	Command is forwarded to the central server for ticket invalidation.

Limitations and notes

- Helix Swarm is not supported with the centralized authentication server.
- All servers that use `P4AUTH` must have the same Unicode setting as the central authorization server.
- Setting `P4AUTH` by means of a `p4 configure set P4AUTH=[protocol:]server:port` command requires a restart of the outer server.

  If you need to set `P4AUTH` for a replica, use the following syntax:

  ```
  p4 configure set ServerName#P4AUTH=[protocol:]server:port
  ```
- If you have set `P4AUTH`, no warning will be given if you delete a user who has an open file or client.
- To ensure that `p4 review` and `p4 reviews` work correctly, you must enable remote depot access for the service user (or, if no service user is specified, for a user named `remote`) on the central server.

  Note: There is no `remote` type user but there is a special user named `remote` that is used to define protections for a remote depot.
- To ensure that the authentication server correctly distinguishes forwarded commands from commands issued by trusted, directly-connected users, you must define any IP-based protection entries in the Perforce service by prepending the string “proxy-” to the `[protocol:]host:port` definition.

  > **Important**
  > Before you prepend the string `proxy-` to the workstation’s IP address, make sure that a broker or proxy is in place.
- Protections for non-forwarded commands are enforced by the outer server and use the plain client IP address, even if the protections are derived from lines in the central server’s protections table.

Centralized changelist server (P4CHANGE)

By default, Helix servers do not coordinate the numbering of changelists. Each Helix Core server numbers its changelists independently. If you are running multiple servers, you can configure your servers to refer to a *centralized changelist server* from which to obtain changelist numbers. Doing so ensures that changelist numbers are unique across your organization, regardless of the server to which they are submitted.

> **Note**
> When using a centralized changelist server, all outer servers must be at the same (or newer) release level as the central server.

To configure Helix server to use a centralized changelist server, set `P4CHANGE` before starting the second server, or specify it on the `p4d` command line with the `-g` option:

```
$ p4d -g central:1666 -p 1999
```

> **Note**
> On Windows, configure the outer server with `p4 set -S` as follows:
>
> ```
> C:\> p4 set -S "Outer Server" P4CHANGE=central:1666
> C:\> p4 set -S "Outer Server" P4PORT=1999
> ```

In this example, the outer server is configured to use a centralized changelist server (named `central`). Whenever a user of the outer server must assign a changelist number (that is, when a user creates a pending changelist or submits one), the centralized server's next available changelist number is used instead.

There is no limit on the number of servers that can refer to a centralized changelist server. This configuration has no effect on the output of the `p4 changes` command; `p4 changes` lists only changelists from the *currently* connected server, regardless of whether it generates its own changelist numbers or relies on a centralized changelist server.

If your server is making use of a centralized changelist server, the following line will appear in the output of `p4 info`:

```
...
Changelist Server: [protocol:]host:port
```

Where `[protocol:]host:port` refers to the protocol, host, and port number of the centralized changelist server.

Helix Broker

This topic assumes you have read "Deployment architecture" on page 369.

The work needed to install and configure a broker is minimal: the administrator needs to configure the broker and configure the users to access the Helix server through the broker. Broker configuration involves the use of a configuration file that contains rules for specifying which commands individual users can execute and how commands are to be redirected to the appropriate Perforce service. You do not need to back up the broker. In case of failure, you just need to restart it and make sure that its configuration file has not been corrupted.

From the perspective of the end user, the broker is transparent: users connect to a Helix Broker just as they would connect to any other Helix Core server.

> **Note**
> Historically, brokers were the only means to offload 'read only' traffic from a master server to a replica. We now recommend using forwarding-replica or forwarding-standby servers for such offloading because they handle it automatically.

System requirements

To use the Helix Broker, you must have:

- A Helix server (`p4d`) at release 2007.2 or higher (2012.1 or higher to use SSL).
- Helix server applications at release 2007.2 or higher (2012.1 or higher to use SSL).

The Helix Broker is designed to run on a host that lies close to the Helix server, preferably on the same machine.

Installing the broker

Non-package-based installation of the Broker

1. Download the `p4broker` executable from the Perforce website at https://www.perforce.com/downloads/helix-broker-p4broker
2. Copy the download to a suitable directory on the host (such as `/usr/local/bin`), and ensure that the binary is executable:

```
$ chmod +x p4broker
```

Linux package-based installation of the Broker

This topic assumes you have met the "Prerequisites" on page 35 of "Linux package-based installation" on page 35.

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

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

The `helix-broker` package installs the main component of the Broker, `p4broker`, as well as the command line interface (`p4`), the service controller (`p4dctl`), and a configuration script to set them up.

Package installation requires sudo or root level privileges.

Verify the Public Key

To ensure you have the correct public key for installing Perforce packages, verify the fingerprint of the Perforce public key against the fingerprint shown below.

1. Download the public key at https://package.perforce.com/perforce.pubkey
2. To obtain the fingerprint of the public key, run one of the following:

for Ubuntu 18.04/20.04 and RHEL/CentOS 8
`gpg -n --import --import-options import-show perforce.pubkey`

for Ubuntu 14.04/16.04 and RHEL/CentOS 6/7
`gpg --with-fingerprint perforce.pubkey`

3. Verify that it matches this fingerprint:
   `E581 31C0 AEA7 B082 C6DC 4C93 7123 CB76 0FF1 8869`

Follow the instructions that apply to you:

- "For APT (Ubuntu) " on the facing page
- "For YUM (Red Hat Enterprise Linux or CentOS)" on the facing page
- "For SUSE Linux Enterprise Server" on the facing page

For APT (Ubuntu)

1. Add the Perforce packaging key to your APT keyring
   ```
   wget -qO - https://package.perforce.com/perforce.pubkey | sudo
   apt-key add -
   ```
2. Add the Perforce repository to your APT configuration.

   Create a file called `/etc/apt/sources.list.d/perforce.list` with the following line:

   ```
   deb http://package.perforce.com/apt/ubuntu {distro} release
   ```

   Where `{distro}` is replaced by one of the following: `precise`, `trusty`, `xenial` or `bionic`.
3. Run `apt-get update`
4. Install the package by running `sudo apt-get install helix-broker`

You can also browse the repository and download a Deb file directly from https://package.perforce.com/apt/

For YUM (Red Hat Enterprise Linux or CentOS)

1. Add Perforce's packaging key to your RPM keyring:
   ```
   sudo rpm --import https://package.perforce.com/perforce.pubkey
   ```
2. Add Perforce's repository to your YUM configuration.

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

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

   where `{`*`version`*`}` is either 6 for RHEL 6 or 7 for RHEL 7
3. Install the package by running `sudo yum install broker`

- You can also browse the repository and download an RPM file directly: https://package.perforce.com/yum/

For SUSE Linux Enterprise Server

1. Add Perforce's packaging key to your RPM keyring:
   ```
   sudo rpm --import http://package.perforce.com/perforce.pubkey
   ```
2. Add the Perforce repository.
   ```
   sudo zypper addrepo http://package.perforce.com/yum/rhel/7/x86_
   64/ helix
   ```

3. Install the package by running `sudo zypper install broker`

■ You can also browse the repository and download an RPM file directly: https://package.perforce.com/yum/

Running the broker

After you have created your configuration file, start the Helix Broker from the command line by issuing the following command:

```
$ p4broker -c config_file
```

Alternatively, you can set `P4BROKEROPTIONS` before launching the broker and use it to specify the broker configuration file (or other options) to use.

For example, on Unix:

```
$ export P4BROKEROPTIONS="-c /usr/perforce/broker.conf"
$ p4broker -d
```

and on Windows:

```
C:\> p4 set -s P4BROKEROPTIONS="-c c:\p4broker\broker.conf"
C:\> p4broker
```

The Helix Broker reads the specified broker configuration file, and on Unix platforms the `-d` option causes the Helix Broker to detach itself from the controlling terminal and run in the background.

To configure the Helix Broker to start automatically, create a startup script that sets `P4BROKEROPTIONS` and runs the appropriate `p4broker` command.

On Windows systems, you can also set `P4BROKEROPTIONS` and run the broker as a service. This involves the following steps:

```
C:\> cd C:\p4broker\
C:\p4broker\> copy p4broker.exe p4brokers.exe
C:\p4broker\> copy "C:\Program Files\Perforce\Server\svcinst.exe"
svcinst.exe
C:\p4broker\> svcinst create -n P4Broker -e
"C:\p4broker\p4brokers.exe" -a
C:\p4broker\> p4 set -S P4Broker P4BROKEROPTIONS="-c
C:\p4broker\p4broker.conf"
C:\p4broker\> svcinst start -n P4Broker
```

`svcinst.exe` is a standard Windows program. ***`P4Broker`*** is the name given to the Windows service. For more information, see the Knowledge Base article, "Installing P4Broker on Windows and Unix systems".

Enabling SSL support

To encrypt the connection between a Helix Broker and its end users, your broker must have a valid private key and certificate pair in the directory specified by its **`P4SSLDIR`** environment variable. Certificate and key generation and management for the broker works the same as it does for the Helix Core server. The users' Helix server applications must be configured to trust the fingerprint of the broker.

To encrypt the connection between a Helix Broker and a Helix Core server, your broker must be configured so as to trust the fingerprint of the Helix Core server. That is, the user that runs **`p4broker`** must create a `P4TRUST` file (using `p4 trust`) that recognizes the fingerprint of the Helix Core server, and must set **`P4TRUST`**, specifying the path to that file (**`P4TRUST`** cannot be specified in the broker configuration file).

For more information about enabling SSL for the broker, see the Support Knowledgebase article, "Enabling SSL Support for the Server/Broker/Proxy ".

Broker information

You can issue the **`p4 info`** to determine whether you are connected to a broker or not. When connected to a broker, the **`Broker address`** and **`Broker version`** appear in the output:

```
$ p4 info
User name: bruno
Client name: bruno-ws
Client host: bruno.host
Client root: /Users/bruno/Workspaces/depot
Current directory: /Users/bruno/Workspaces/depot/main/jam
Peer address: 192.168.1.40:55138
Client address: 192.168.1.114
Server address: perforce:1667
Server root: /perforce/server/root
Server date: 2014/03/13 15:46:52 -0700 PDT
Server uptime: 92:26:02
Server version: P4D/LINUX26X86_64/2014.1/773873 (2014/01/21)
ServerID: master-1666
Broker address: perforce:1666 Broker version: P4BROKER/LINUX26X86_
64/2014.1/782990
Server license: 10000 users (support ends 2016/01/01)
Server license-ip: 192.168.1.40
Case Handling: sensitive
```

When connected to a broker, you can use the **`p4 broker`** command to see a concise report of the broker's info:

```
$ p4 broker
Current directory: /Users/bruno/Workspaces/depot/main/jam
Client address: 192.168.1.114:65463
Broker address: perforce:1666
Broker version: P4BROKER/LINUX26X86_64/2014.1/782990
```

Broker and protections

To apply the IP address of a broker user's workstation against the protections table, prepend the string **`proxy-`** to the workstation's IP address.

> **Important**
> Before you prepend the string **`proxy-`** to the workstation's IP address, make sure that a broker or proxy is in place.

Consider an organization with a remote development site with workstations on a subnet of **`192.168.10.0/24`**. The organization also has a central office where local development takes place; the central office exists on the **`10.0.0.0/8`** subnet. A Perforce service resides in the **`10.0.0.0/8`** subnet, and a broker resides in the **`192.168.10.0/24`** subnet. Users at the remote site belong to the group **`remotedev`**, and occasionally visit the central office. Each subnet also has a corresponding set of IPv6 addresses.

To ensure that members of the **`remotedev`** group use the broker while working at the remote site, but do not use the broker when visiting the local site, add the following lines to your protections table:

```
list    group    remotedev     192.168.10.0/24              -//...
list    group    remotedev     [2001:db8:16:81::]/48        -//...

write   group    remotedev     proxy-192.168.10.0/24         //...
write   group    remotedev     proxy-[2001:db8:16:81::]/48   //...

list    group    remotedev     proxy-10.0.0.0/8             -//...
list    group    remotedev     proxy-[2001:db8:1008::]/32   -//...

write   group    remotedev     10.0.0.0/8                    //...
write   group    remotedev     [2001:db8:1008::]/32          //...
```

The first line denies `list` access to all users in the `remotedev` group if they attempt to access Helix server without using the broker from their workstations in the `192.168.10.0/24` subnet. The second line denies access in identical fashion when access is attempted from the IPV6 `[2001:db8:16:81::]/48` subnet.

The third line grants `write` access to all users in the `remotedev` group if they are using the broker and are working from the `192.168.10.0/24` subnet. Users of workstations at the remote site must use the broker. (The broker itself does not have to be in this subnet, for example, it could be at `192.168.20.0`.) The fourth line grants access in identical fashion when access is attempted from the IPV6 `[2001:db8:16:81::]/48` subnet.

Similarly, the fifth and sixth lines deny `list` access to `remotedev` users when they attempt to use the broker from workstations on the central office's subnets (`10.0.0.0/8` and `[2001:db8:1008::]/32`). The seventh and eighth lines grant write access to `remotedev` users who access the Helix server directly from workstations on the central office's subnets. When visiting the local site, users from the `remotedev` group must access the Helix server directly.

When the Perforce service evaluates protections table entries, the `dm.proxy.protects` configurable is also evaluated.

`dm.proxy.protects` defaults to `1`, which causes the `proxy-` prefix to be prepended to all client host addresses that connect via an intermediary (proxy, broker, broker, or edge server), indicating that the connection is not direct.

Setting `dm.proxy.protects` to `0` removes the `proxy-` prefix and allows you to write a single set of protection entries that apply both to directly-connected clients as well as to those that connect via an intermediary. This is more convenient but less secure if it matters that a connection is made using an intermediary. If you use this setting, all intermediaries must be at release 2012.1 or higher.

P4Broker options

Option	Meaning
`-c` *`file`*	Specify a configuration file. Overrides `P4BROKEROPTIONS` setting.
`-C`	Output a sample configuration file, and then exit.
`-d`	Run as a daemon (in the background).
`-f`	Run as a single-threaded (non-forking) process.
`-h`	Print help message, and then exit.
`-q`	Run quietly (no startup messages).
`-V`	Print broker version, and then exit.

Option	Meaning
`-v subsystem=level`	Set server trace options. Overrides the value of the `P4DEBUG` setting, but does *not* override the `debug-level` setting in the `p4broker.conf` file. Default is null. The server command trace options and their meanings are as follows. ▪ `server=0` Disable broker command logging. ▪ `server=1` Logs broker commands to the server log file. ▪ `server=2` In addition to data logged at level `1`, logs broker 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.
`-Gc`	Generate SSL credentials files for the broker: create a private key (`privatekey.txt`) and certificate file (`certificate.txt`) 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 `config.txt` is present in `P4SSLDIR`, generate a self-signed certificate with specified characteristics.
`-Gf`	Display the fingerprint of the broker'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.

Configuring the broker

P4Broker is controlled by a broker configuration file. The broker configuration file is a text file that contains rules for:

- Specifying which commands that individual users can use.
- Defining commands that are to be redirected to a specified replica server.

To generate a sample broker configuration file, issue the following command:

```
$ p4broker -C > p4broker.conf
```

You can edit the newly created `p4broker.conf` file to specify your requirements.

Format of broker configuration files

A broker configuration file contains the following sections:

- Global settings: settings that apply to all broker operations
- Alternate server definitions: the addresses and names of replica servers to which commands can be redirected in specified circumstances
- Command handler specifications: specify how individual commands should be handled. In the absence of a command handler for any given command, the Helix Broker permits the execution of that command.

Next step

"Specifying hosts" below

Specifying hosts

The broker configuration requires specification of the `target` setting, which identifies the Perforce service to which commands are to be sent, the `listen` address, which identifies the address where the broker listens for commands from Helix server client applications, and the optional `altserver` alternate server address, which identifies a replica, proxy, or other broker connected to the Perforce service.

The host specification uses the format ***`protocol`***`:`***`host`***`:`***`port`***, where ***`protocol`*** is the communications protocol (beginning with `ssl:` for SSL, or `tcp:` for plaintext), ***`host`*** is the name or IP address of the machine to connect to, and ***`port`*** is the number of the port on the host.

Protocol	Behavior
`<not set>`	If the `net.rfc3484` configurable is set, allow the OS to determine which transport is used. This is applicable only if a host name (either FQDN or unqualified) is used. If an IPv4 literal address (for example, `127.0.0.1`) is used, the transport is always `tcp4`, and if an IPv6 literal address (for example, `::1`) is used, then the transport is always `tcp6`.

Protocol	Behavior
`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 on/connect to an IPv4 address/port. If this fails, try IPv6.
`tcp64:`	Attempt to listen on/connect to an IPv6 address/port. 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:`	Attempt to listen on/connect to an IPv4 address/port. If this fails, try IPv6. After connecting, require SSL encryption.
`ssl64:`	Attempt to listen on/connect to an IPv6 address/port. If this fails, try IPv4. After connecting, require SSL encryption.

The ***`host`*** field can be the hosts' hostname or its IP address; both IPv4 and IPv6 addresses are supported. For the `listen` setting, you can use the `*` wildcard to refer to all IP addresses, but only when you are not using CIDR notation.

If you use the `*` wildcard with an IPv6 address, you must enclose the entire IPv6 address in square brackets. For example, `[2001:db8:1:2:*]` is equivalent to `[2001:db8:1:2::]/64`. Best practice is to use CIDR notation, surround IPv6 addresses with square brackets, and to avoid the `*` wildcard.

Next step

"Global settings" below

Global settings

The following settings apply to all operations you specify for the broker.

Setting	Meaning	Example
`target`	The default Helix Core server (P4D) to which commands are sent unless overridden by other settings in the configuration file.	`target = [protocol:]host:port;`
`listen`	The address on which the Helix Broker listens for commands from Helix server client applications.	`listen = [protocol:][host:]port;`
`directory`	The home directory for the Helix Broker. Other paths specified in the broker configuration file must be relative to this location.	`directory = path`;
`logfile`	Path to the Helix Broker logfile.	`logfile = path;`
`debug-level`	Level of debugging output to log. Overrides the value specified by the `-v` option and `P4DEBUG`. You can specify the value for one or more flags.	`debug-level = server=1, time=1, rpl=3;`
`admin-name`	The name of your Helix server Administrator. This is displayed in certain error messages.	`admin-name = "P4 Admin";`
`admin-email`	An email address where users can contact their Helix server Administrator. This address is displayed to users when broker configuration problems occur.	`admin-email = admin@example.com;`
`admin-phone`	The telephone number of the Helix server Administrator.	`admin-phone = nnnnnnn;`
`redirection`	The redirection mode to use: `selective` or `pedantic`. In `selective` mode, redirection is permitted within a session until one command has been executed against the default (target) server. From then on, all commands within that session run against the default server and are not redirected. In `pedantic` mode, all requests for redirection are honored. The default mode is `selective`.	`redirection = selective;`

Setting	Meaning	Example
`service-user`	An optional user account by which the broker authenticates itself when communicating with a target server. The broker configuration does not include a setting for specifying a password as this is considered insecure. Use the `p4 -u` ***`service-user`*** `login -p` command to generate a ticket. Store the displayed ticket value in a file, and then set the `ticket-file` setting to the path of that file. To provide continuous operation of the broker, the `service-user` user should be included in a group that has its `Timeout` setting set to `unlimited`. The default ticket timeout is 12 hours.	`service-user = svcbroker;`
`ticket-file`	An optional alternate location for `P4TICKETS` files.	`ticket-file = /home/p4broker/.p4tickets;`
`compress`	Compress connection between broker and server. Over a slow link such as a WAN, compression can increase performance. If the broker and the server are near to each other (and especially if they reside on the same physical machine), then bandwidth is not an issue, and compression should be disabled to spare CPU cycles.	`compress = false;`

Setting	Meaning	Example

`altserver`	An optional alternate server to help reduce the load on the target server.
	The ***`name`*** assigned to the alternate server is used in command handler specifications.
	See "Alternate server definitions" on page 480.
	The syntax is:
	`altserver: name { target=[protocol:]host:port; }`
	Multiple **`altserver`** settings may appear in the broker configuration file, one for each alternate server. For example:
	`altserver: rep_18310 {`
	`target=10.5.10.118:18310; }`
	`altserver: rep_18320 {`
	`target=10.5.10.118:18320; }`
	`altserver: rep_18330 {`
	`target=10.5.10.118:18330; }`	

Next step

"Command handler specifications" below

Command handler specifications

Command handlers enable you to specify how the broker responds to different commands issued by different users from within different environments. When users run commands, the Helix Broker searches for matching command handlers and uses the first match found. If no command handler matches the user's command, the command is forwarded to the target Helix Core server for normal processing.

The general syntax of a command handler specification is outlined in the sample **`broker.conf`**:

```
command: commandpattern
{
# Conditions for the command to meet (optional)
# Note that with the exception of 'flags', these are regex patterns.
  flags           = required-flags;
  args            = required-arguments;
  user            = required-user;
  workspace       = required-client-workspace;
  prog            = required-client-program;
  version         = required-version-of-client-program;

  # What to do with matching commands (required)
  action  = pass | reject | redirect | filter | respond ;

  # How to go about it
  destination = altserver;            # Required for action = redirect
  execute = /path/to/filter/program;  # Required for action = filter
  message = rejection-message;        # Required for action = reject
}
```

The ***commandpattern*** parameter can be a regular expression and can include the **.*** wildcard. For example, a ***commandpattern*** of **user.*** matches both the **p4 user** and **p4 users** commands. See "Regular expression synopsis" on the facing page.

The following table describes the parameters in detail.

Parameter	Meaning
flags	A list of options that must be present on the command line of the command being handled. This feature enables you to specify different handling for the same **p4** command, depending on which options the user specifies. Note that only single character options may be specified here. Multi-character options, and options that take arguments should be handled by a filter program.
args	A list of arguments that must be present on the command line of the command being handled.
user	The name of the user who issued the command.
workspace	The Helix server client workspace setting in effect when the command was issued.
prog	The Helix server client application through which the user issued the command. This feature enables you to handle commands on a per-application basis.
version	The version of the Helix server application through which the user issued the command.
action	Defines how the Helix Broker handles the specified commands. Valid values are: **pass**, **reject**, **redirect**, **filter**, or **respond**.

Parameter	Meaning
`destination`	For redirected commands, the name of the replica to which the commands are redirected. The destination must be the name of a previously defined alternate (replica) server listed in the `altserver` setting. You can implement load-balancing by setting the destination to the keyword `random`. Commands are randomly redirected to any alternate (replica) server that you have already defined. You can also set destination to the `address:port` of the server where you want commands redirected.
`execute`	The path to a filter program to be executed. For details about filter programs, see "Filter programs" on the next page.
`message`	A message to be sent to the user, typically before the command is executed; this may be used with any of the above actions.
`checkauth`	Authenticates the connection. If set to `true`, the Helix Broker checks that the user has access to the Helix Core server before performing the action by running `p4 protects -m` with the user's connection. If set to `false`, or if not set, Helix Broker does not perform the check. If a filter program is run, the highest level permission that the user has is passed in as the `maxPerm` parameter. For details about filter programs, see "Filter programs" on the next page.

For example, the following command handler prevents user `joe` from invoking `p4 submit` from the `buildonly` client workspace.

```
command: submit
{
    user = joe;
    workspace = buildonly;
    action = reject;
    message = "Submit failed: Please do not submit from this
workspace."
}
```

Regular expression synopsis

A regular expression, or *regex*, is a sequence of characters that forms a search pattern, for use in pattern matching with strings. The following is a short synopsis of the regex facility available in command handler specifications.

A regular expression is formed from zero or more *branches*. Branches are separated by `|`. The regex matches any string that matches at least one of the branches.

A branch is formed from zero or more *pieces*, concatenated together. A branch matches when all of its pieces match in sequence, that is, a match for the first piece, followed by a match for the second piece, and so on.

A piece is an *atom* possibly followed by a *quantifier*: `*`, `+`, or `?`. An atom followed by `*` matches a sequence of 0 or more instances of the atom. An atom followed by `+` matches a sequence of 1 or more instances of the atom. An atom followed by `?` matches a sequence of 0 or 1 instances of the atom.

An atom is:

- a subordinate regular expression in parentheses - matches that subordinate regular expression
- a range (see below),
- `.` - matches any single character,
- `^` - matches the beginning of the string,
- `$` - matches the end of the string,
- a `\` followed by a single character - matches that character,
- or a single character with no other significance - matches that character.

A range is a sequence of characters enclosed in square brackets (`[]`), and normally matches any single character from the sequence. If the sequence begins with `^`, it matches any single character that is *not* in the sequence. If two characters in the sequence are separated by `-`, this is shorthand for the full list of ASCII characters between them (for example, `[0-9]` matches any decimal digit, `[a-z]` matches any lowercase alphabetical character). To include a literal `]` in the sequence, make it the first character (following a possible `^`). To include a literal `-`, make it the first or last character.

Filter programs

When the ***`action`*** for a command handler is `filter`, the Helix Broker executes the program or script specified by the **`execute`** parameter and performs the action returned by the program. Filter programs enable you to enforce policies beyond the capabilities provided by the broker configuration file.

The Helix Broker invokes the filter program by passing command details to the program's standard input in the following format:

Command detail	Definition
`command:`	User command
`brokerListenPort:`	Port on which the broker is listening
`brokerTargetPort:`	Port on which the target server is listening
`clientPort:`	P4PORT setting of the client
`clientProg:`	Client application program

Command detail	Definition
`clientVersion:`	Version of client application program
`clientProtocol:`	Level of client protocol
`apiProtocol:`	Level of api protocol
`maxLockTime:`	Maximum lock time (in ms) to lock tables before aborting
`maxPerm`	Highest permission (if "checkauth" on page 476 is set)
`maxResults:`	Maximum number of rows of result data to be returned
`maxScanRows:`	Maximum number of rows of data scanned by a command
`workspace:`	Name of client workspace
`user:`	Name of requesting user
`clientIp:`	IP address of client
`proxyIp:`	IP address of proxy (if any)
`cwd:`	Client's working directory
`argCount:`	Number of arguments to command
`Arg0:`	First argument (if any)
`Arg1:`	Second argument (if any)
`clientHost:`	Hostname of the client
`brokerLevel:`	The internal version level of the broker.
`proxyLevel:`	The internal version level of the proxy (if any).

Non-printable characters in command arguments are sent to filter programs as a percent sign followed by a pair of hex characters representing the ASCII code for the non-printable character in question. For example, the tab character is encoded as `%09`.

Your filter program must read this data from STDIN before performing any additional processing, regardless of whether the script requires the data. If the filter script does not read the data from STDIN, "broken pipe" errors can occur, and the broker rejects the user's command.

Your filter program must respond to the Broker on standard output (stdout) with data in one of the four following formats:

```
action: PASS
message: a message for the user (optional)
```

```
action: REJECT
message: a message for the user (required)
```

```
action: REDIRECT
altserver: (an alternate server name)
message: a message for the user (optional)
```

```
action: RESPOND
message: a message for the user (required)
```

```
action: CONTINUE
```

> **Note**
> The values for the `action` are case-sensitive.

The `action` keyword is always required and tells the Broker how to respond to the user’s request. The available `action`s are:

Action	Definition
`PASS`	Run the user’s command unchanged. A `message` for the user is optional.
`REJECT`	Reject the user’s command; return an error message. A `message` for the user is required.
`REDIRECT`	Redirect the command to a different (alternate) replica server. An `altserver` is required. See "Configuring alternate servers to work with central authorization servers" on the facing page for details. A `message` for the user is optional. To implement this action, the broker makes a new connection to the alternate server and routes all messages from the client to the alternate server rather than to the original server. This is unlike HTTP redirection where the client is requested to make its own direct connection to an alternate web server.
`RESPOND`	Do not run the command; return an informational message. A `message` for the user is required.
`CONTINUE`	Defer to the next command handler matching a given command. For information on using multiple handlers, see the Support Knowledgebase article, "How the Broker can process multiple command handlers".

If the filter program returns any response other than something complying with the four message formats above, the user’s command is rejected. If errors occur during the execution of your filter script code cause the broker to reject the user’s command, the broker returns an error message.

Broker filter programs have difficulty handling multi-line message responses. You must use syntax like the following to have new lines be interpreted correctly when sent from the broker:

```
message="\"line 1\nline 3\nline f\n\""
```

That is, the string must be quoted twice.

Next step

"Alternate server definitions" below

Alternate server definitions

The Helix Broker can direct user requests to an alternate server to reduce the load on the target server. These alternate servers must be replicas (or brokers, or proxies) connected to the intended target server.

To set up and configure a replica server, see "Replication" on page 379. The broker works with both metadata-only replicas and with replicas that have access to both metadata and versioned files.

You can define any number of alternate replica servers in a broker configuration file.

The syntax for specifying an alternate server is:

```
altserver: name { target=[protocol:]host:port; }
```

For example:

```
altserver: rep_18310 { target=10.5.10.118:18310; }
altserver: rep_18320 { target=10.5.10.118:18320; }
altserver: rep_18330 { target=10.5.10.118:18330; }
```

The name you assign to the alternate server is used in "Command handler specifications" on page 474.

Configuring alternate servers to work with central authorization servers

Alternate servers require users to authenticate themselves when they run commands. For this reason, the Helix Broker must be used in conjunction with a central authorization server (**P4AUTH**). For more information about setting up a central authorization server, see "Authorization and changelist servers" on page 457.

When used with a central authorization server, a single `p4 login` request can create a ticket that is valid for the user across all servers in the Helix Broker’s configuration, enabling the user to log in once. The Helix Broker assumes that a ticket granted by the target server is valid across all alternate servers.

If the target server in the broker configuration file is a central authorization server, the value assigned to the **target** parameter must precisely match the setting of `P4AUTH` on the alternate server machine (s). Similarly, if an alternate sever defined in the broker configuration file is used as the central authorization server, the value assigned to the **target** parameter for the alternate server must match the setting of **P4AUTH** on the other server machine(s).

Example of a P4Broker configuration file

If you choose to use a broker instead of a forwarding replica, edit the P4Broker configuration file to add the **target**, **listen** port, and other broker information.

```
target      = master:11111;
listen      = 33333;
```

```
directory   = /p4broker/root/;
logfile     = broker.log;
debug-level = server=1;
admin-name  = "your name";
admin-phone = x1234;
admin-email = your.name@yourcompany.dom
redirection = selective;

#
# Add an "altserver" for the replica:
#
altserver: replica1
{
    target  = replica:22222;
}

#
# Add command handlers to redirect read-only metadata commands to the
replica
#
command: ^(branches|changes|clients|counters|depots|dirs \
    |filelog|files|fstat|groups|interchanges|jobs|labels|opened \
    |sizes|fixes|where|workspaces|users)$
{
    action  = redirect;
    destination = replica1;
}

#
# Prevent user joe from invoking p4 submit from the
# buildonly client workspace.
#
command: submit
{
    user = joe;
    workspace = buildonly;
    action = reject;
    message = "Submit failed: Please do not submit from this
workspace.";
}

#
# Allow user 'maria' to run 'p4 opened -a' but not without the '-a'
option
#
command: opened
{
    flags = -a;
    user = maria;
    action = pass;
}
command: opened
{
    user = maria;
    action = reject;
    message = "Please use 'p4 opened -a'";
}
```

Helix Proxy

This topic assumes you have read "Deployment architecture" on page 369.

To improve performance obtained by multiple Helix server users accessing a shared Helix server repository across a WAN,

1. Configure P4P on the side of the network close to the users.
2. Configure the users to access the service through P4P.
3. Configure P4P to access the master Perforce service.

System requirements

To use Helix Proxy, you must have:

- Helix server release 2002.2 or later (2012.1 or later to use SSL)
- Sufficient disk space on the proxy host to store a cache of file revisions

Installing P4P

In addition to the basic steps described next, see:

- "Using SSL to encrypt connections to a Helix server" on page 123
- "Defending from man-in-the-middle attacks" on page 488

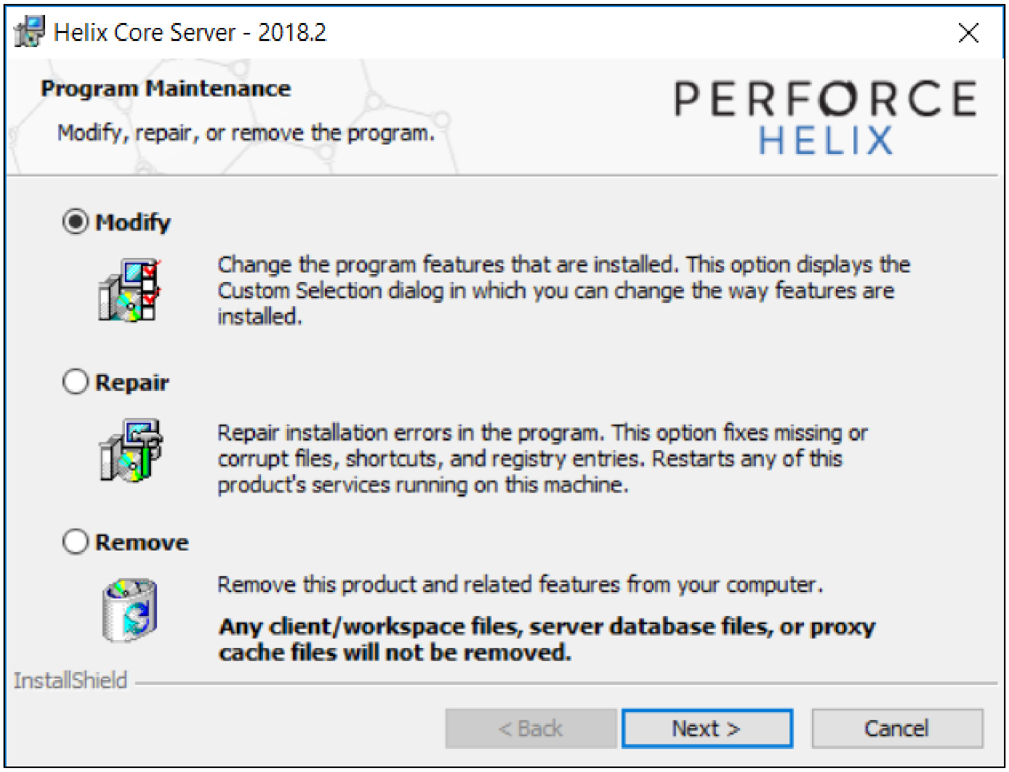
UNIX

To install P4P on UNIX or Linux, do the following:

1. Download the **`p4p`** executable to the machine on which you want to run the proxy.
2. Select a directory on this machine (`P4PCACHE`) in which to cache file revisions.
3. Select a port (`P4PORT`) on which **`p4p`** will listen for requests from Helix server applications.
4. Select the target Helix server (`P4TARGET`) for which this proxy will cache.

Windows

Install P4P as an option when running the Helix Core server installer for Windows:



Helix Core Server - 2018.2

Program Maintenance

Modify, repair, or remove the program.

● **Modify**

Change the program features that are installed. This option displays the Custom Selection dialog in which you can change the way features are installed.

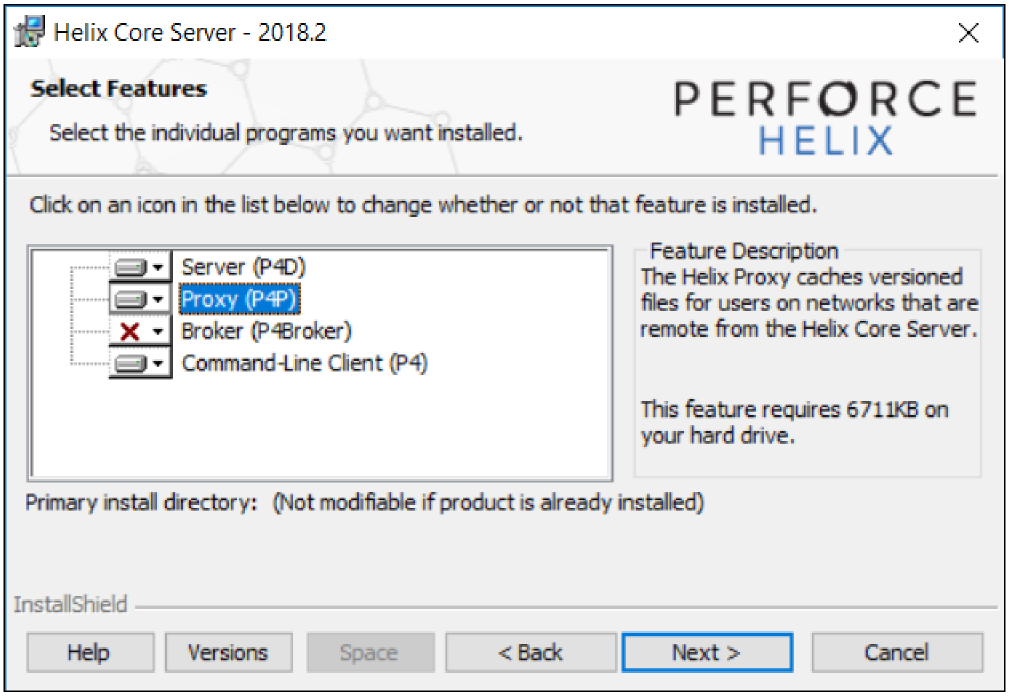
○ **Repair**

Repair installation errors in the program. This option fixes missing or corrupt files, shortcuts, and registry entries. Restarts any of this product's services running on this machine.

○ **Remove**

Remove this product and related features from your computer.

Any client/workspace files, server database files, or proxy cache files will not be removed.



Helix Core Server - 2018.2

Select Features

Select the individual programs you want installed.

Click on an icon in the list below to change whether or not that feature is installed.

- Server (P4D)
- Proxy (P4P)
- ☒ Broker (P4Broker)
- Command-Line Client (P4)

Feature Description

The Helix Proxy caches versioned files for users on networks that are remote from the Helix Core Server.

This feature requires 6711KB on your hard drive.

Primary install directory: (Not modifiable if product is already installed)

Running P4P

To run Helix Proxy, invoke the `p4p` executable, configuring it with environment variables or command-line options. Options you specify on the command line override environment variable settings.

For example, the following command line starts a proxy that communicates with a central Helix server located on a host named `central`, listening on port 1666.

```
$ p4p -p tcp64:[::]:1999 -t central:1666 -r /var/proxyroot
```

To use the proxy, Helix server applications connect to P4P on port 1999 on the machine where the proxy runs. The proxy listens on both the IPv6 and IPv4 transports. P4P file revisions are stored under a directory named `/var/proxyroot`.

P4P supports connectivity over IPv6 networks as well as IPv4. See P4PORT in *Helix Core P4 Command Reference*.

Running P4P as a Windows service

To run P4P as a Windows service, either install P4P from the Windows installer, or specify the `-s` option when you invoke `p4p.exe`, or rename the P4P executable to `p4ps.exe`.

To pass parameters to the P4Proxy service, set the `P4POPTIONS` registry variable using the `p4 set` command. For example, if you normally run the Proxy with the command:

```
C:\> p4p -p 1999 -t ssl:mainserver:1666
```

You can set the **P4POPTIONS** variable for a Windows service named **Helix Proxy** by setting the service parameters as follows:

```
C:\> p4 set -S "Perforce Proxy" P4POPTIONS="-p 1999 -t
ssl:mainserver:1666"
```

When the **"Helix Proxy"** service starts, P4P listens for plaintext connections on port 1999 and communicates with the Helix Core server via SSL at **ssl:mainserver:1666**.

P4P options

The following command-line options specific to the proxy are supported:

Proxy options

Option	Meaning
-d	Run as daemon - fork first, then run (UNIX only).
-f	Do not fork - run as a single-threaded server (UNIX only).
-i	Run for **inetd** (socket on **stdin/stdout** - UNIX only).
-q	Run quietly; suppress startup messages.
-c	Do not compress data stream between the Helix server to P4P. (This option reduces CPU load on the central server at the expense of slightly higher bandwidth consumption.)
-s	Run as a Windows service (Windows only). Running **p4p.exe -s** is equivalent to invoking **p4ps.exe**.
-S	Disable cache fault coordination. The proxy maintains a table of concurrent sync operations, called **pdb.lbr**, to avoid multiple transfers of the same file. This mechanism prevents unnecessary network traffic, but can impart some delay to operations until the file transfer is complete. When **-S** is used, cache fault coordination is disabled, allowing multiple transfers of files to occur. The proxy then decides whether to transfer a file based solely on its checksum. This may increase the burden on the network, while potentially providing speedier completion for sync operations.

General options

Option	Meaning
`-h` or `-?`	Display a help message.
`-V`	Display the version of the Helix Proxy.
`-r` *`root`*	Specify the directory where revisions are cached. Default is `P4PCACHE`, or the directory from which **`p4p`** is started if **`P4PCACHE`** is not set.
`-R` *`root`*	Specify the directory where the proxy database is stored. See `P4PROOT`.
`-L` *`logfile`*	Specify the location of the log file. Default is `P4LOG`, or the directory from which **`p4p`** is started if **`P4LOG`** is not set.
`-p` *`port`*	Specify the port on which P4P will listen for requests from Helix server applications. Default is `P4PORT`, or 1666 if **`P4PORT`** is not set.
`-t` *`port`*	Specify the port of the target Helix server (that is, the Helix server for which P4P acts as a proxy). Default is `P4TARGET` or **`perforce:1666`** if **`P4TARGET`** is not set.
`-e` *`size`*	Cache only those files that are larger than ***`size`*** bytes. Default is **`P4PFSIZE`**, or zero (cache all files) if **`P4PFSIZE`** is not set.
`-u` *`serviceuser`*	For proxy servers, authenticate as the specified **`serviceuser`** when communicating with the central server. The service user must have a valid ticket before the proxy will work.
`-v` *`level`*	Specifies server trace level. Debug messages are stored in the proxy server's log file. Debug messages from **`p4p`** are not passed through to **`p4d`**, and debug messages from **`p4d`** are not passed through to **`p4p`** proxies. Default is **`P4DEBUG`**, or none if **`P4DEBUG`** is not set.

Certificate-handling options

Option	Meaning
-Gc	Generate SSL credentials files for the proxy: create a private key (**`privatekey.txt`**) and certificate file (**`certificate.txt`**) 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 **`config.txt`** is present in **`P4SSLDIR`**, generate a self-signed certificate with specified characteristics.

Option	Meaning
-Gf	Display the fingerprint of the proxy'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.

Proxy monitoring options

Option	Meaning
`-l`	List pending archive transfers
`-l-s`	List pending archive transfers, summarized
`-v lbr.stat.interval=`*`n`*	Set the file status interval in seconds.
`-v proxy.monitor.level=`*`n`*	`0`: (default) Monitoring disabled `1`: Proxy monitors file transfers only `2`: Proxy monitors all operations `3`: Proxy monitors all traffic for all operations
`-v proxy.monitor.interval=`*`n`*	Proxy monitoring interval, in seconds. If not set, defaults to 10 seconds.

`-m1`
`-m2`
`-m3`
	Show currently-active connections and their status.
	Requires **`proxy.monitor.level`** set equal to or greater than 1. The optional argument specifies the level of detail: **`-m1`**, **`-m2`**, or **`-m3`** show increasing levels of detail corresponding to the **`proxy.monitor.level`** setting.

Proxy archive cache options

See the lbr.proxy.case configurable in *Helix Core P4 Command Reference*.

Administering P4P

The following sections describe the tasks involved in administering a proxy.

No backups required

You never need to back up the P4P cache directory.

If necessary, P4P reconstructs the cache based on Helix server metadata.

Stopping P4P

P4P is effectively stateless; to stop P4P under UNIX, `kill` the `p4p` process with `SIGTERM` or `SIGKILL`. Under Windows, click **End Process** in the **Task Manager**.

Upgrading P4P

After you have replaced the `p4p` executable with the upgraded version, you must also remove the `pdb.lbr` and `pdb.monitor` files (if they exist) from the proxy root before you restart the upgraded proxy.

Enabling SSL support

To encrypt the connection between a Helix Proxy and its end users, your proxy must have a valid private key and certificate pair in the directory specified by its `P4SSLDIR` environment variable. Certificate and key generation and management for the proxy works the same as it does for the Helix Core server. See "Using SSL to encrypt connections to a Helix server" on page 123. The users' Helix server applications must be configured to trust the fingerprint of the proxy.

To encrypt the connection between a Helix Proxy and its upstream Perforce service, your proxy installation must be configured to trust the fingerprint of the upstream Perforce service. That is, the user that runs `p4p` (typically a service user) must create a `P4TRUST` file (using `p4 trust`) that recognizes the fingerprint of the upstream Perforce service.

See the Knowledge Base article, "Enabling SSL Support for the Server/Broker/Proxy".

Defending from man-in-the-middle attacks

You can use the `net.mimcheck` configurable to enable checks for possible interception or modification of data. These settings are pertinent for proxy administration:

- A value of 3 checks connections from clients, proxies, and brokers for TCP forwarding.
- A value of 5 requires that proxies, brokers, and all Helix server intermediate servers have valid logged-in service users associated with them. This allows administrators to prevent unauthorized proxies and services from being used.

You must restart the server after changing the value of this configurable. See `net.mimcheck`.

Localizing P4P

If your Helix server has localized error messages (see "Localizing server error messages" on page 86), you can localize your proxy's error message output by shutting down the proxy, copying the server's `db.message` file into the proxy root, and restarting the proxy.

Managing disk space consumption

P4P caches file revisions in its cache directory. These revisions accumulate until you delete them. P4P does not delete its cached files or otherwise manage its consumption of disk space.

> **Warning**
> If you do not delete cached files, you will eventually run out of disk space. To recover disk space, remove files under the proxy's root.
>
> Although you do not need to stop the proxy to delete its cached files or the `pdb.lbr` file, removing the cache or `pdb.lbr` file is NOT recommended during a sync operation because it might cause the following error: "Proxy could not update its cache".

If you delete files from the cache without stopping the proxy, you must also delete the `pdb.lbr` file at the proxy's root directory. (The proxy uses the `pdb.lbr` file to keep track of which files are scheduled for transfer, so that if multiple users simultaneously request the same file, only one copy of the file is transferred.)

Determining if your Helix server applications are using the proxy

If your Helix server application is using the proxy, the proxy's version information appears in the output of `p4 info`.

For example, if a Perforce service is hosted at `ssl:central:1666` and you direct your Helix server application to a Helix Proxy hosted at `outpost:1999`, the output of `p4 info` resembles the following:

```
$ export P4PORT=tcp:outpost:1999
$ p4 info
User name: p4adm
Client name: admin-temp
Client host: remotesite22
Client root: /home/p4adm/tmp
Current directory: /home/p4adm/tmp
Client address: 192.168.0.123
Server address: central:1666
Server root: /usr/depot/p4d
Server date: 2012/03/28 15:03:05 -0700 PDT
Server uptime: 752:41:23
Server version: P4D/FREEBSD4/2012.1/406375 (2012/01/25)
Server encryption: encrypted
Proxy version: P4P/SOLARIS26/2012.1/406884 (2012/01/25)
Server license: P4 Admin <p4adm> 20 users (expires 2013/01/01)
Server license-ip: 10.0.0.2
Case handling: sensitive
```

P4P and protections

For setting protections on proxies and brokers, see "Proxy and protections" under "Setting protections with p4 protect" on page 148.

Determining if specific files are being delivered from the proxy

Use the `-Zproxyverbose` option with `p4` to display messages indicating whether file revisions are coming from the proxy (`p4p`) or the central server (`p4d`). For example:

```
$ p4 -Zproxyverbose sync noncached.txt
//depot/main/noncached.txt - refreshing /home/p4adm/tmp/noncached.txt
$ p4 -Zproxyverbose sync cached.txt
//depot/main/cached.txt - refreshing /home/p4adm/tmp/cached.txt
File /home/p4adm/tmp/cached.txt delivered from proxy server
```

Case-sensitivity issues and the proxy

If you are running the proxy on a case-sensitive platform such as UNIX, and your users are submitting files from case-insensitive platforms (such as Windows), the default behavior of the proxy is to fold case. For example, `FILE.TXT` can overwrite `File.txt` or `file.txt`.

In the case of text files and source code, the performance impact of this behavior is negligible. If, however, you are dealing with large binaries such as `.ISO` images or `.VOB` video objects, there can be performance issues associated with this behavior.

After any change to `lbr.proxy.case`, you must clear the cache before restarting the proxy.

Maximizing performance improvement

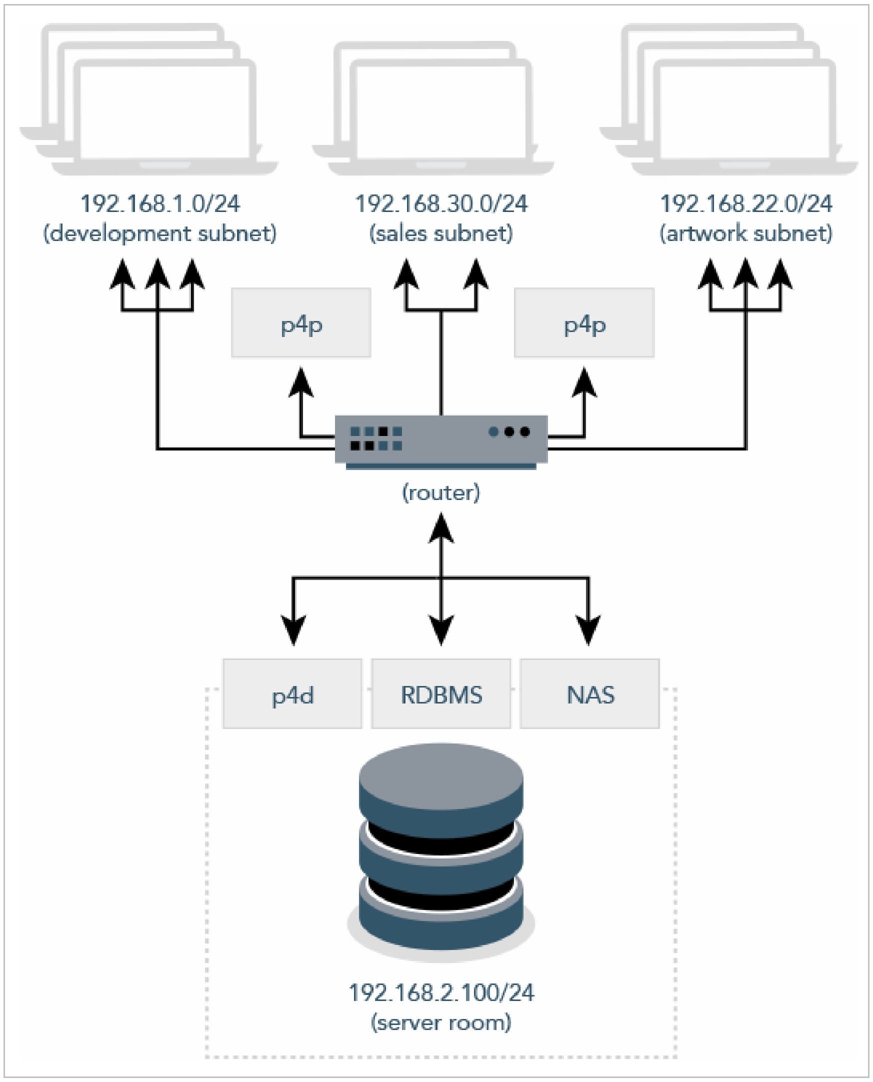
In addition to the topics in this chapter, see the Support Knowledgebase article on tuning tips for "Proxy Performance", including how to minimize the syncing of small files.

Reducing server CPU usage by disabling file compression

By default, P4P compresses communication between itself and the Helix server versioning service, imposing additional overhead on the service. To disable compression, specify the `-c` option when you invoke `p4p`. This option is particularly effective if you have excess network and disk capacity and are storing large numbers of binary file revisions in the depot, because the proxy (rather than the upstream versioning service) decompresses the binary files from its cache before sending them to Helix server users.

Network topologies versus P4P

If network bandwidth on the subnet with the Perforce service is nearly saturated, deploy the proxies on the other side of a router so that the traffic from end users to the proxy is isolated to a subnet separate from the subnet containing the Perforce service. You might split the subnet into multiple subnets and deploy a proxy in each resulting subnet:



Preloading the cache directory for optimal initial performance

Helix Proxy stores file revisions only when one of its users submits a new revision to the depot or requests an existing revision from the depot. That is, file revisions are not prefetched. Performance gains from P4P occur only after file revisions are cached.

After starting P4P, you can prefetch the cache directory by creating a dedicated client workspace and syncing it to the head revision. All other users who subsequently connect to the proxy immediately obtain the performance improvements provided by P4P. For example, a development site located in Asia with a P4P server targeting a Helix server in North America can preload its cache directory by using an automated job that runs a `p4 sync` against the entire Helix server depot after most work at the North American site has been completed, but before its own developers arrive for work.

By default, `p4 sync` writes files to the client workspace. If you have a dedicated client workspace that you use to prefetch files for the proxy, however, this step is redundant. If this machine has slower I/O performance than the machine running the Helix Proxy, it can also be time-consuming.

To preload the proxy's cache without the redundant step of also writing the files to the client workspace, use the `-Zproxyload` option when syncing. For example:

```
$ export P4CLIENT=prefetch
$ p4 sync //depot/main/written.txt
//depot/main/written.txt - refreshing /home/prefetch/main/written.txt
$ p4 -Zproxyload sync //depot/main/nonwritten.txt
//depot/main/nonwritten.txt - file(s) up-to-date.
```

Both files are now cached, but `nonwritten.txt` is never written to the the `prefetch` client workspace. When prefetching the entire depot, the time savings can be considerable.

Distributing disk space consumption

P4P stores revisions as if there were only one depot tree. If this approach stores too much file data onto one filesystem, you can use symbolic links to spread the revisions across multiple filesystems.

For instance, if the P4P cache root is `/disk1/proxy`, and the Helix server it supports has two depots named `//depot` and `//released`, you can split data across disks, storing `//depot` on `disk1` and `//released` on `disk2` as follows:

```
$ mkdir /disk2/proxy/released
$ cd /disk1/proxy
$ ln -s /disk2/proxy/released released
```

The symbolic link means that when P4P attempts to cache files in the `//released` depot to `/disk1/proxy/released`, the files are stored on `/disk2/proxy/released`.

Backing up and upgrading services

Backing up and upgrading services in a multi-server environment involve special considerations.

Backing up services

How you back up a service in your multi-server deployment depends upon the service type:

Broker	■ stores no data locally ■ back up its configuration file manually
Proxy	■ requires no backups and automatically rebuilds its cache of data if files are missing ■ contains no logic to detect when diskspace is running low. Periodically monitor your proxy to ensure it has sufficient diskspace.
Server	■ Follow the backup procedures described at "Backup and recovery" on page 175. • If you are using an edge-commit architecture, both the commit server and the edge servers must be backed up. See "Backup and recovery planning" on page 454. ■ Backup requirements for replicas that are not edge servers vary depending on your site's requirements. ■ Consider taking checkpoints offline so that your users are not blocked from accessing the primary server during lengthy checkpoint operations. See "Taking Checkpoints on Edge and Replica Servers" below. ■ Although a checkpoint (`p4d -jc`) is NOT supported on an edge or replica server, you CAN take a checkpoint dump on an edge or replica server (`p4d -jd`). See the Helix Core server (p4d) Reference. ■ Maintaining journals: • on edge servers is a best practice • on replica servers is optional, and you can disable such journals by using `p4d -J off` ■ You can have triggers fire when the journal is rotated on an edge or replica server. See "Triggering on journal rotation" on page 316. ■ Journal rotation on a replica or edge server begins AFTER the master has completed its journal rotation

Taking Checkpoints on Edge and Replica Servers

First, run `p4 admin checkpoint` against the edge or replica:

```
p4 -p edge:1666 admin checkpoint -Z
```

The background journal **`pull`** command will perform the checkpoint at the next rotation of the journal on the master.

This results in a message about the scheduling of the checkpoint and a file called **`stateCKP`** being written to the edge or replica server root (P4ROOT) directory containing information about the scheduling of the checkpoint. For example:

```
Checkpoint scheduled at 1472141783 (2020/03/26 09:16:23 -
0700 PDT ); opts:
```

To cancel a scheduled checkpoint, remove the **`stateCKP`** file from the edge or replica **`P4ROOT`** prior to rotating the journal on the commit or master server.

Second, run **`p4 admin journal`** against the commit or master:

```
p4 -p commit:1666 admin journal
Rotating journal to journal.40...
```

> **Note**
> Do not use the **`-z`** flag to **`p4 admin journal`** or **`p4d -jj`**
>
> This is because rotated commit and master server journals initially need to be uncompressed. Otherwise replication could be adversely affected.

Detecting Coordinated Checkpoint Completion

To determine that a coordinated checkpoint has completed, record the journal counter on the commit or master at the time the edge or replica checkpoint is scheduled. For example, the following **`counter`** command, run against your commit or master server, reports the current value of the journal counter on that server. The **`admin checkpoint`** command, run on the edge or replica, schedules a checkpoint on that server the next time a journal rotation is detected on the master.

```
p4 -p commit:1666 counter journal
40
p4 -p edge:1666 admin checkpoint -Z
The 'pull' command will perform the checkpoint at the next rotation
of the journal on the master.
```

In the example above, the journal counter is reported as **`40`**, which means that the next checkpoint will be **`41`**. To find out whether the checkpoint has completed, use one of the following.

Checkpoint Checksum

When a checkpoint completes, an md5 checksum of the checkpoint contents is written alongside the checkpoint:

```
$ ls -l edge1/checkpoint.41*
-r--r--r-- 1 bruno staff 11833462 Aug 25 09:59 checkpoint.41
-r--r--r-- 1 bruno staff 55 Aug 25 09:59 checkpoint.41.md5
```

Look for the writing of the md5 checksum, which means the checkpoint has completed.

A `journal-rotate` trigger on the edge or replica

Configure a **`journal-rotate`** trigger on the edge or replica. This fires when the edge or replica journal is rotated. Since journal rotation is a sign of a successful checkpoint, if the trigger fires you know the checkpoint has completed. See "Triggering on journal rotation" on page 316.

Checkpoint History

The p4 journals command displays information from the `db.ckphist` table which holds historical information about checkpoint and journal activity. For example, you can report on the last checkpoint taken using:

```
p4 journals -F type=checkpoint -m1
... start 1472142210
... startDate 2018/08/25 09:23:30
... end 1472142211
... endDate 2018/08/25 09:23:31
... pid 53536
... type checkpoint
... flags -q true (admin checkpoint)
... jnum 40
... jfile checkpoint.40
... jdate 1472142211
... jdateDate 2018/08/25 09:23:31
... jdigest 7A5080F52EC13518305AD2A93919864A
... jsize 11833462
... jtype text
```

Once a checkpoint has been scheduled and you know the checkpoint sequence number of the next edge or replica checkpoint, poll the edge or replica using p4 journals for the next checkpoint:

```
p4 journals -F 'type=checkpoint jnum=41'
```

The command returns without providing any output until the checkpoint has completed, at which time you'll see the details of the checkpoint completion in the p4 journals output:

```
p4 journals -F 'type=checkpoint jnum=41'
... start 1472144358
... startDate 2018/08/25 09:59:18
... end 1472144358
... endDate 2018/08/25 09:59:18
... pid 53757
... type checkpoint
... flags -q true (admin checkpoint)
... jnum 41
... jfile checkpoint.41
... jdate 1472144358
... jdateDate 2018/08/25 09:59:18
... jdigest 22971CDC1E26C70B1E6A58C92C4820AA
... jsize 11833460
... jtype text
```

Note

If a checkpoint fails, the p4 journals output contains information about the failure, including the error message related to the failure. For example:

```
p4 journals -m1
... start 1452184543
... startDate 2018/01/07 08:35:43
... end 1452184543
... endDate 2018/01/07 08:35:43
... pid 98622
... type checkpoint
... flags  (admin checkpoint)
... jnum 41
... jfile /Volumes/backups/checkpoint.41
... jdate 1452184543
... jdateDate 2018/01/07 08:35:43
... jdigest CFF44FD4B9B26AD90F93AC71D4E47418
... jsize 65536
... jtype text
... failed 1
... errmsg write: /Volumes/backups/checkpoint.41: No space left on
device
```

Upgrading services

Servers, brokers, and proxies must be at the same release level. When upgrading:

1. Shut down the furthest-upstream service or commit server and permit the system to quiesce.
2. Upgrade downstream services first, starting with the replica that is furthest downstream, working

upstream towards the master or commit server.

3. Keep downstream services stopped until the server immediately upstream has been upgraded.

> **Tip**
> See the instructions on upgrading at "Upgrading the server" on page 56.

Helix Core server (p4d) Reference

Start the Perforce service, perform checkpoint/journaling, or do certain system administration tasks.

Syntax

```
p4d [ options ]
p4d.exe [ options ]
p4s.exe [ options ]
p4d -j? [ -z | -Z ] [ args ... ]
```

Description

p4d [*options*] **p4d.exe** [*options*] **p4s.exe** [*options*]	invoke the background process that manages the Helix server versioning service
p4d -j? [-z | -Z] [*args* ...]	for certain system administration tasks, including some that are related to checkpointing and journaling

> **Note**
> Rotating the journal means saving the existing journal and creating a new, empty journal for future transactions.
>
> "Truncating" a journal refers to the new journal file starting out as an empty file.

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

This section includes the following types of options: "Server options" below, "General options" on the facing page, "Checkpointing options" on page 502, "Journal restore options" on page 505, "Replication and multi-server options" on page 506, "Journal dump and restore filtering" on page 507, "Certificate handling" on page 508, and "Configuration options" on page 508.

Server 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).
`-n`	Start the server in maintenance mode, which means that: ▪ the server is only able to perform commands that do not require a client ▪ the user and file count restrictions listed in the license file are not enforced
`--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`.
`--daemonsafe`	Is like `-d` and forks the `p4d` into the background, but also closes the stdio (standard input output) files.
`-xi`	Irreversibly reconfigure the Helix Core server (and its metadata) to operate in Unicode mode. Do not use this option unless you know you require Unicode mode. For details, see the *Release Notes* and the *Internationalization Notes*.

Server options	Meaning
`-xu`	Run database upgrades and quit. Upgrades must be run manually unless the server is a DVCS personal server, which runs upgrade steps automatically.
`-xv`	Run low-level database validation and quit. ■ With no table arguments, validates all tables. ■ If one or more table arguments are provided, only the specified tables are validated. For example: `p4d -xv db.rev db.change db.have`
`-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.
`-xU CleanServerLocks`	Safely cleans up the server locks directory, which can help avoid the possible issue of running out of inodes. (See also the Support Knowledgebase article about possible causes of the message No space left on device.)

General options

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 displayed by the `p4 -z tag info` command. Overrides `P4DESCRIPTION` setting.
`-In` *name*	A server name for use with `p4 configure`. Overrides `P4NAME` setting. Although this is supported for backward compatibility, an easier method for configuring the server is to use p4 serverid and p4 server.
`-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`.

General options	Meaning
`-r root`	Specify the server root directory. Overrides `P4ROOT`. Default is current working directory.
`-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`.
`--show-realtime`	Display real-time monitoring values on the command line. For details of this feature, see `p4 monitor realtime` in *Helix Core P4 Command Reference*.

Checkpointing options

Checkpointing options	Meaning
`-c command`	Lock database tables, run ***`command`***, unlock the tables, and exit.
`-jc [ prefix ]`	Journal-create; create checkpoint and `.md5` file, and rotates the journal. Rotating the journal means saving the existing journal and creating a new, empty journal for future transactions. 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.

Checkpointing options	Meaning
`-jd` *`file`*	Journal-checkpoint; create checkpoint and `.md5` file. No journal rotation occurs.
`-z -jd` *`file`*	same as -jd except the checkpoint is compressed
`-jj [` *`prefix`* `]`	Rotates journal, and no checkpointing occurs.
`-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.
`-z -jr` *`file`*	Journal-restore; restore metadata from a **compressed** checkpoint and/or journal file. If you specify the `-r $P4ROOT` option on the command line, the `-r` option must precede the `-jr` option.

Checkpointing options	Meaning
`-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 is 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. 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, Helix server automatically adds the `.gz` extension to the checkpoint file. So, the command: `p4d -jd -z myCheckpoint` creates two files: `myCheckpoint.gz` and `myCheckpoint.md5`. **Warning** If you have downstream replicas, use `-Z` instead of `-z` because they cannot read from a compressed journal.

Checkpointing options	Meaning
`-Z`	Compress (in `gzip` format) checkpoint, but leave journal uncompressed for use by replica servers. That is, it applies to `-jc`, not `-jd`. **Note** Can be used when taking a checkpoint that rotates the journal: `p4d -Z -jc` The `-Z` option is not used for recovery: `p4d -jr`

Journal restore options

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 replica servers started with the `p4 replicate` command.

Journal restore options	Meaning
`-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 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 Support.

Replication and multi-server options

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 and restore filtering

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

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 **`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

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 Helix server 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

- Do not run p4d as root. See "Running the Helix server (p4d) as an unprivileged user" on page 46.

- 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 Helix server 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 Helix server database.
- If your users use triggers, don't use the `-f` (non-forking mode) option. To run trigger scripts, the Perforce service needs to be able to "fork" (spawn copies of itself).
- 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. For details, see the *Release Notes*.

Typical tasks

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 compress checkpoints and journals, which creates two files: `myCheckpoint.gz` and `myCheckpoint.md5`.	`p4d -jd -z myCheckpoint`
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`
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`

Helix Core Server Control (p4dctl)

The Helix Core Server Control (`p4dctl`) utility enables the management of Perforce services running on the local host.

The root user:

- is the Linux owner of the `/etc/perforce/` directory, and can start and stop all services
- can configure the `/etc/perforce/p4dctl.conf` file to allow one or more non-root users, such as the `perforce` user, to start and stop certain services

> **Note**
> `p4dctl` can only be obtained as part of a Linux package installation.

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, `p4dctl.conf`. This prevents failures that stem from the differences between the user’s environment and that of `root`.

> **Warning**
> Helix environment variables:
>
> - must be defined in the P4DCTL configuration file
> - will NOT take effect if they are defined from the `perforce` user's shell environment, such as the `.bashrc` file

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 process identifier (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.

> **Note**
> If privileges, ownership, or configuration is incorrect, the user will see the following:
>
> **p4dctl error:**
>
> **'master' p4d: '/opt/perforce/sbin/p4d -p 1666' exited with status 255.**
>
> which is also recorded in the log.

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**, such as checkpointing and journal rotation
- the environment in which managed services are running

  The environment is configured using environment variables that can 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 516.

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 can also contain comments. A comment is 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 might be inside or outside of a server block.

- 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:

- logging is enabled
- the server logging level is set to **`1`**
- the correct **`P4CONFIG`** files are used
- the p4dctl MAINTENANCE script (introduced in the 2019.1 release) runs every night at 04:00. This is merely a checkpoint, not a complete backup solution.

```
Environment
{
    P4LOG = log
    P4DEBUG = "server=1" # Embedded = requires quotes
    P4CONFIG = .p4config
    MAINTENANCE = true
}
```

Note
Common settings can be placed into the `/etc/perforce/p4dctl.conf` configuration file.

- The `/etc/perforce/p4dctl.conf` settings can be overridden by setting them in configuration files inside the P4DCTL directory. By default, `/etc/perforce/p4dctl.conf.d` is the directory for such configuration files.
- Any configurables for a server that were made using `p4 configure set` or in the server spec override the P4DCTL settings.

Server block

A server block defines settings and variables that apply only to the specified type of service:

Type	Meaning
`p4d`	Helix Core server, also called Helix server
`p4p`	Helix Proxy
`p4broker`	P4Broker
`p4ftp`	P4FTP plugin
`p4web`	Helix server 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 single command to stop p4d, proxy, and p4ftp 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`.
	The string *must* be quoted. For example:
	`Args = "-C1"`
	or
	`Args = "-u us_proxy -v lbr.stat.interval=300 -v proxy.monitor.level=3 -v proxy.monitor.interval=300"`

`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

Setting	Meaning

`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 define the environment variables:

Type	Variable	Setting
`p4d`	**`P4PORT`**	Port to use for this service
	`P4ROOT`	Path to the server's root directory
	`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

Type	Variable	Setting
`p4ftp`	`PORT`	Address of the target Perforce service
	`P4FTPPORT`	Port to use for serving FTP requests
`p4web`	`PORT`	Address of the target Helix 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 Helix Core 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 with the **`.conf`** extension.

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" on page 15

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 512.

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.

Glossary

A

access level

A permission assigned to a user to control which commands the user can execute. See also the 'protections' entry in this glossary and the 'p4 protect' command in the P4 Command Reference.

admin access

An access level that gives the user permission to privileged commands, usually super privileges.

APC

The Alternative PHP Cache, a free, open, and robust framework for caching and optimizing PHP intermediate code.

archive

1. For replication, versioned files (as opposed to database metadata). 2. For the 'p4 archive' command, a special depot in which to copy the server data (versioned files and metadata).

atomic change transaction

Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.

avatar

A visual representation of a Swarm user or group. Avatars are used in Swarm to show involvement in or ownership of projects, groups, changelists, reviews, comments, etc. See also the "Gravatar" entry in this glossary.

B

base

For files: The file revision that contains the most common edits or changes among the file revisions in the source file and target file paths. For checked out streams: The public have version from which the checked out version is derived.

binary file type

A Helix server file type assigned to a non-text file. By default, the contents of each revision are stored in full, and file revision is stored in compressed format.

branch

(noun) A set of related files that exist at a specific location in the Perforce depot as a result of being copied to that location, as opposed to being added to that location. A group of related files is often referred to as a codeline. (verb) To create a codeline by copying another codeline with the 'p4 integrate', 'p4 copy', or 'p4 populate' command.

branch form

The form that appears when you use the 'p4 branch' command to create or modify a branch specification.

branch mapping

Specifies how a branch is to be created or integrated by defining the location, the files, and the exclusions of the original codeline and the target codeline. The branch mapping is used by the integration process to create and update branches.

branch view

A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. This is the same as branch mapping.

broker

Helix Broker, a server process that intercepts commands to the Helix server and is able to run scripts on the commands before sending them to the Helix server.

C

change review

The process of sending email to users who have registered their interest in changelists that include specified files in the depot.

changelist

A list of files, their version numbers, the changes made to the files, and a description of the changes made. A changelist is the basic unit of versioned work in Helix server. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot. See also atomic change transaction and changelist number.

changelist form

The form that appears when you modify a changelist using the 'p4 change' command.

changelist number

An integer that identifies a changelist. Submitted changelist numbers are ordinal (increasing), but not necessarily consecutive. For example, 103, 105, 108, 109. A pending changelist number might be assigned a different value upon submission.

check in

To submit a file to the Helix server depot.

check out

To designate one or more files, or a stream, for edit.

checkpoint

A backup copy of the underlying metadata at a particular moment in time. A checkpoint can recreate db.user, db.protect, and other db.* files. See also metadata.

classic depot

A repository of Helix server files that is not streams-based. Uses the Perforce file revision model, not the graph model. The default depot name is depot. See also default depot, stream depot, and graph depot.

client form

The form you use to define a client workspace, such as with the 'p4 client' or 'p4 workspace' commands.

client name

A name that uniquely identifies the current client workspace. Client workspaces, labels, and branch specifications cannot share the same name.

client root

The topmost (root) directory of a client workspace. If two or more client workspaces are located on one machine, they should not share a client root directory.

client side

The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.

client workspace

Directories on your machine where you work on file revisions that are managed by Helix server. By default, this name is set to the name of the machine on which your client workspace is located, but it can be overridden. Client workspaces, labels, and branch specifications cannot share the same name.

code review

A process in Helix Swarm by which other developers can see your code, provide feedback, and approve or reject your changes.

codeline

A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

comment

Feedback provided in Helix Swarm on a changelist, review, job, or a file within a changelist or review.

commit server

A server that is part of an edge/commit system that processes submitted files (checkins), global workspaces, and promoted shelves.

conflict

1. A situation where two users open the same file for edit. One user submits the file, after which the other user cannot submit unless the file is resolved. 2. A resolve where the same line is changed when merging one file into another. This type of conflict occurs when the comparison of two files to a base yields different results, indicating that the files have been changed in different ways. In this case, the merge cannot be done automatically and must be resolved manually. See file conflict.

copy up

A Helix server best practice to copy (and not merge) changes from less stable lines to more stable lines. See also merge.

counter

A numeric variable used to track variables such as changelists, checkpoints, and reviews.

CSRF

Cross-Site Request Forgery, a form of web-based attack that exploits the trust that a site has in a user's web browser.

D

default changelist

The changelist used by a file add, edit, or delete, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.

deleted file

In Helix server, a file with its head revision marked as deleted. Older revisions of the file are still available. in Helix server, a deleted file is simply another revision of the file.

delta

The differences between two files.

depot

A file repository hosted on the server. A depot is the top-level unit of storage for versioned files (depot files or source files) within a Helix Core server. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

The left side of any client view mapping, specifying the location of files in a depot.

depot syntax

Helix server syntax for specifying the location of files in the depot. Depot syntax begins with: //depot/

diff

(noun) A set of lines that do not match when two files, or stream versions, are compared. A conflict is a pair of unequal diffs between each of two files and a base, or between two versions of a stream. (verb) To compare the contents of files or file revisions, or of stream versions. See also conflict.

donor file

The file from which changes are taken when propagating changes from one file to another.

E

edge server

A replica server that is part of an edge/commit system that is able to process most read/write commands, including 'p4 integrate', and also deliver versioned files (depot files).

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

extension

Similar to a trigger, but more modern. See "Helix Core Server Administrator Guide" on "Extensions".

F

file conflict

In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also, an attempt to submit a file that is not an edit of the head revision of the file in the depot, which typically occurs when another user opens the file for edit after you have opened the file for edit.

file pattern

Helix server command line syntax that enables you to specify files using wildcards.

file repository

The master copy of all files, which is shared by all users. In Helix server, this is called the depot.

file revision

A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, preceded by a pound sign (#), for example testfile#3.

file tree

All the subdirectories and files under a given root directory.

file type

An attribute that determines how Helix server stores and diffs a particular file. Examples of file types are text and binary.

fix

A job that has been closed in a changelist.

form

A screen displayed by certain Helix server commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

A replica server that can process read-only commands and deliver versioned files (depot files). One or more replicate servers can significantly improve performance by offloading some of the master server load. In many cases, a forwarding replica can become a disaster recovery server.

G

Git Fusion

A Perforce product that integrates Git with Helix, offering enterprise-ready Git repository management, and workflows that allow Git and Helix server users to collaborate on the same projects using their preferred tools.

graph depot

A depot of type graph that is used to store Git repos in the Helix server. See also Helix4Git and classic depot.

group

A feature in Helix server that makes it easier to manage permissions for multiple users.

H

have list

The list of file revisions currently in the client workspace.

head revision

The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.

heartbeat

A process that allows one server to monitor another server, such as a standby server monitoring the master server (see the p4 heartbeat command).

Helix server

The Helix server depot and metadata; also, the program that manages the depot and metadata, also called Helix Core server.

Helix TeamHub

A Perforce management platform for code and artifact repository. TeamHub offers built-in support for Git, SVN, Mercurial, Maven, and more.

Helix4Git

Perforce solution for teams using Git. Helix4Git offers both speed and scalability and supports hybrid environments consisting of Git repositories and 'classic' Helix server depots.

hybrid workspace

A workspace that maps to files stored in a depot of the classic Perforce file revision model as well as to files stored in a repo of the graph model associated with git.

I

iconv

A PHP extension that performs character set conversion, and is an interface to the GNU libiconv library.

integrate

To compare two sets of files (for example, two codeline branches) and determine which changes in one set apply to the other, determine if the changes have already been propagated, and propagate any outstanding changes from one set to another.

J

job

A user-defined unit of work tracked by Helix server. The job template determines what information is tracked. The template can be modified by the Helix server system administrator. A job describes work to be done, such as a bug fix. Associating a job with a changelist records which changes fixed the bug.

job daemon

A program that checks the Helix server machine daily to determine if any jobs are open. If so, the daemon sends an email message to interested users, informing them the number of jobs in each category, the severity of each job, and more.

job specification

A form describing the fields and possible values for each job stored in the Helix server machine.

job view

A syntax used for searching Helix server jobs.

journal

A file containing a record of every change made to the Helix server's metadata since the time of the last checkpoint. This file grows as each Helix server transaction is logged. The file should be automatically truncated and renamed into a numbered journal when a checkpoint is taken.

journal rotation

The process of renaming the current journal to a numbered journal file.

journaling

The process of recording changes made to the Helix server’s metadata.

L

label

A named list of user-specified file revisions.

label view

The view that specifies which filenames in the depot can be stored in a particular label.

lazy copy

A method used by Helix server to make internal copies of files without duplicating file content in the depot. A lazy copy points to the original versioned file (depot file). Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

license file

A file that ensures that the number of Helix server users on your site does not exceed the number for which you have paid.

list access

A protection level that enables you to run reporting commands but prevents access to the contents of files.

local depot

Any depot located on the currently specified Helix server.

local syntax

The syntax for specifying a filename that is specific to an operating system.

lock

1. A file lock that prevents other clients from submitting the locked file. Files are unlocked with the 'p4 unlock' command or by submitting the changelist that contains the locked file. 2. A database lock that prevents another process from modifying the database db.* file.

log

Error output from the Helix server. To specify a log file, set the P4LOG environment variable or use the p4d -L flag when starting the service.

M

mapping

A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. See also workspace view, branch view, and label view.

MDS checksum

The method used by Helix server to verify the integrity of versioned files (depot files).

merge

1. To create new files from existing files, preserving their ancestry (branching). 2. To propagate changes from one set of files to another. 3. The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.

merge file

A file generated by the Helix server from two conflicting file revisions.

metadata

The data stored by the Helix server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata is stored in the Perforce database and is separate from the archive files that users submit.

modification time or modtime

The time a file was last changed.

MPM

Multi-Processing Module, a component of the Apache web server that is responsible for binding to network ports, accepting requests, and dispatch operations to handle the request.

N

nonexistent revision

A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the #none revision specifier are examples of nonexistent file revisions.

numbered changelist

A pending changelist to which Helix server has assigned a number.

O

opened file

A file you have checked out in your client workspace as a result of a Helix Core server operation (such as an edit, add, delete, integrate). Opening a file from your operating system file browser is not tracked by Helix Core server.

owner

The Helix server user who created a particular client, branch, or label.

P

p4

1. The Helix Core server command line program. 2. The command you issue to execute commands from the operating system command line.

p4d

The program that runs the Helix server; p4d manages depot files and metadata.

P4PHP

The PHP interface to the Helix API, which enables you to write PHP code that interacts with a Helix server machine.

PECL

PHP Extension Community Library, a library of extensions that can be added to PHP to improve and extend its functionality.

pending changelist

A changelist that has not been submitted.

Perforce

Perforce Software, Inc., a leading provider of enterprise-scale software solutions to technology developers and development operations (“DevOps”) teams requiring productivity, visibility, and scale during all phases of the development lifecycle.

project

In Helix Swarm, a group of Helix server users who are working together on a specific codebase, defined by one or more branches of code, along with options for a job filter, automated test integration, and automated deployment.

protections

The permissions stored in the Helix server’s protections table.

proxy server

A Helix server that stores versioned files. A proxy server does not perform any commands. It serves versioned files to Helix server clients.

R

RCS format

Revision Control System format. Used for storing revisions of text files in versioned files (depot files). RCS format uses reverse delta encoding for file storage. Helix server uses RCS format to store text files. See also reverse delta storage.

read access

A protection level that enables you to read the contents of files managed by Helix server but not make any changes.

remote depot

A depot located on another Helix server accessed by the current Helix server.

replica

A Helix server that contains a full or partial copy of metadata from a master Helix server. Replica servers are typically updated every second to stay synchronized with the master server.

repo

A graph depot contains one or more repos, and each repo contains files from Git users.

reresolve

The process of resolving a file after the file is resolved and before it is submitted.

resolve

The process you use to manage the differences between two revisions of a file, or two versions of a stream. You can choose to resolve file conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes. To resolve stream conflicts, you can choose to accept the public source, accept the checked out target, manually accept changes, or combine path fields of the public and checked out version while accepting all other changes made in the checked out version.

reverse delta storage

The method that Helix server uses to store revisions of text files. Helix server stores the changes between each revision and its previous revision, plus the full text of the head revision.

revert

To discard the changes you have made to a file in the client workspace before a submit.

review access

A special protections level that includes read and list accesses and grants permission to run the p4 review command.

review daemon

A program that periodically checks the Helix server machine to determine if any changelists have been submitted. If so, the daemon sends an email message to users who have subscribed to any of the files included in those changelists, informing them of changes in files they are interested in.

revision number

A number indicating which revision of the file is being referred to, typically designated with a pound sign (#).

revision range

A range of revision numbers for a specified file, specified as the low and high end of the range. For example, myfile#5,7 specifies revisions 5 through 7 of myfile.

revision specification

A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, a revision range, change numbers, label names, date/time specifications, or client names.

RPM

RPM Package Manager. A tool, and package format, for managing the installation, updates, and removal of software packages for Linux distributions such as Red Hat Enterprise Linux, the Fedora Project, and the CentOS Project.

S

server data

The combination of server metadata (the Helix server database) and the depot files (your organization's versioned source code and binary assets).

server root

The topmost directory in which p4d stores its metadata (db.* files) and all versioned files (depot files or source files). To specify the server root, set the P4ROOT environment variable or use the p4d -r flag.

service

In the Helix Core server, the shared versioning service that responds to requests from Helix server client applications. The Helix server (p4d) maintains depot files and metadata describing the files and also tracks the state of client workspaces.

shelve

The process of temporarily storing files in the Helix server without checking in a changelist.

status

For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses. For the 'p4 status' command, by default the files opened and the files that need to be reconciled.

storage record

An entry within the db.storage table to track references to an archive file.

stream

A "branch" with built-in rules that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients. Has structured branching, unlike the free-form branching of a "classic" depot. Uses the Perforce file revision model, not the graph model. See also classic depot and graph depot.

stream hierarchy

The set of parent-to-child relationships between streams in a stream depot.

submit

To send a pending changelist into the Helix server depot for processing.

super access

An access level that gives the user permission to run every Helix server command, including commands that set protections, install triggers, or shut down the service for maintenance.

symlink file type

A Helix server file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.

sync

To copy a file revision (or set of file revisions) from the Helix server depot to a client workspace.

T

target file

The file that receives the changes from the donor file when you integrate changes between two codelines.

text file type

Helix server file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

The revision in the depot with which the client file (your file) is merged when you resolve a file conflict. When you are working with branched files, theirs is the donor file.

three-way merge

The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

trigger

A script that is automatically invoked by Helix server when various conditions are met. (See "Helix Core Server Administrator Guide" on "Triggers".)

two-way merge

The process of combining two file revisions. In a two-way merge, you can see differences between the files.

typemap

A table in Helix server in which you assign file types to files.

U

user

The identifier that Helix server uses to determine who is performing an operation. The three types of users are standard, service, and operator.

V

versioned file

Source files stored in the Helix server depot, including one or more revisions. Also known as an archive file. Versioned files typically use the naming convention 'filenamev' or '1.changelist.gz'.

view

A description of the relationship between two sets of files. See workspace view, label view, branch view.

W

wildcard

A special character used to match other characters in strings. The following wildcards are available in Helix server: * matches anything except a slash; ... matches anything including slashes; %%0 through %%9 is used for parameter substitution in views.

workspace

See client workspace.

workspace view

A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.

write access

A protection level that enables you to run commands that alter the contents of files in the depot. Write access includes read and list accesses.

X

XSS

Cross-Site Scripting, a form of web-based attack that injects malicious code into a user's web browser.

Y

yours

The edited version of a file in your client workspace when you resolve a file. Also, the target file when you integrate a branched file.

License Statements

To get a listing of the third-party software licenses that Helix Core server uses, at the command line, type the `p4 help legal` command.

To get a listing of the third-party software licenses that the local client (such as P4V) uses, at the command line, type the `p4 help -l legal` command.