HelixCore

Helix Core Server Administrator Guide: Multi-Site Deployment

2019.2

November 2019

PERFORCE

www.perforce.com

Contents

How to use this guide

This manual is for administrators installing, configuring, and maintaining multiple interconnected or replicated Perforce services.

This guide assumes familiarity with *Helix Core Server Administrator Guide: Fundamentals*.

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

Syntax conventions

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.
[`-f`]	The enclosed elements are 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*.
element1 | *element2*	Either *element1* or *element2* is required.

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.

Earlier versions of this guide:

- 2019.1
- 2018.2
- 2018.1

What's new in this guide

2019.2

Various bug fixes. See the *Release Notes*.

2019.1 release

- End-users can benefit from the "Background archive transfer for edge server submits" on page 72
- "Edge-to-edge chaining" on page 69: 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 ""Introduction to multi-site deployment architectures" on page 13" and "Filtering metadata during replication or edge-to-edge chaining" on page 69.

2018.2 release

Various bug fixes as mentioned in the *Release Notes*.

2018.1 release

To help the standby server stay as current as possible with the master server, consider using the configurable that enables writing to the device on which the standby server's active journal would be located. See the mention of the `rpl.journalcopy.location` configurable at "Configuring a read-only replica" on page 50.

2017.2 release

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.

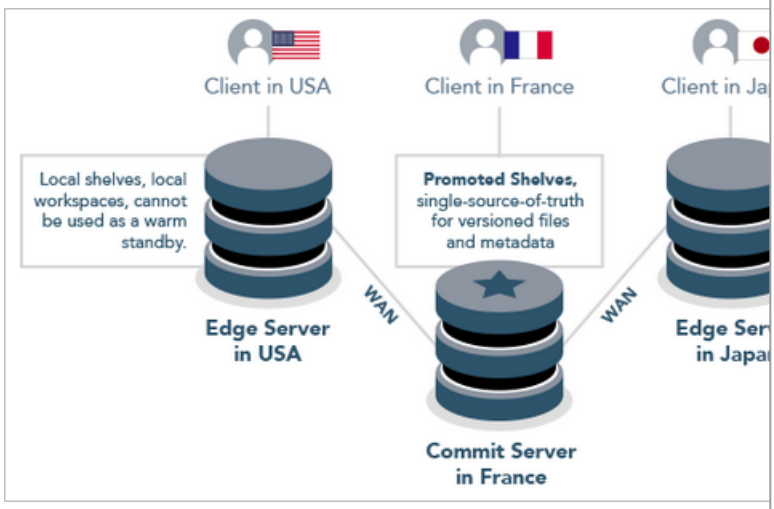
Helix Core server Control (p4dctl) has moved

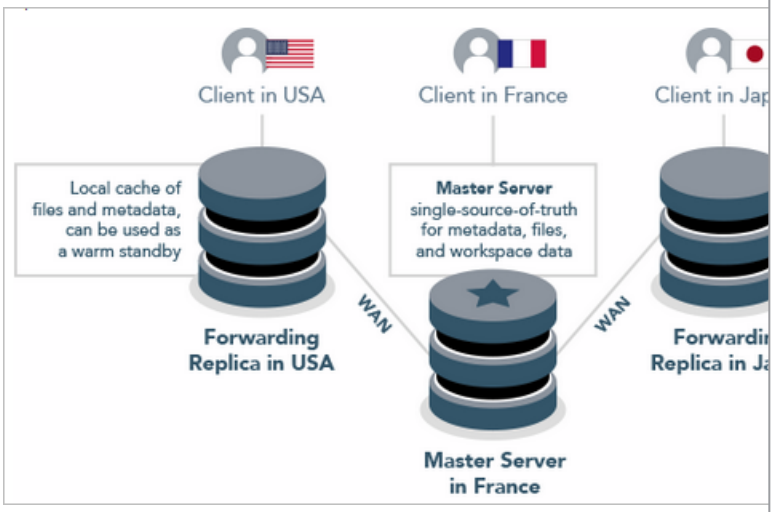
The appendix formerly named **Helix Versioning Engine Control (p4dctl)**, which was both in this guide (volume 2 of the "Helix Versioning Engine Administrator Guide") and in the volume 1, "Helix Versioning Engine Administrator Guide: Fundamentals" (volume 1) is now exclusively in volume 1 at https://www.perforce.com/perforce/doc.current/manuals/p4sag/#P4SAG/appendix.p4dctl.html.

Introduction to multi-site deployment architectures

Helix Core Server Administrator Guide: Fundamentals explains how you create, configure, and maintain a single Helix Core server. 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

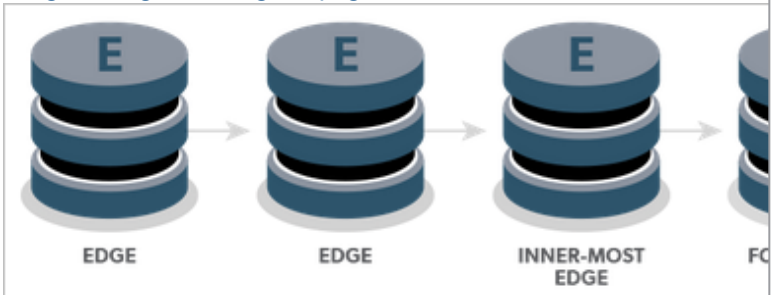
Architecture	Advantage	Disadvantage
"Commit-edge" on page 77 	■ best performance overall because most commands are local ■ an edge server that is used only for automated processing, such as builds, can be deployed without a backup/recovery solution because the edge local data is critical only during build-time.	■ cannot be used as a warm standby ■ requires machine provisioning and administration, including backups of each edge (except in the case of a build edge server)

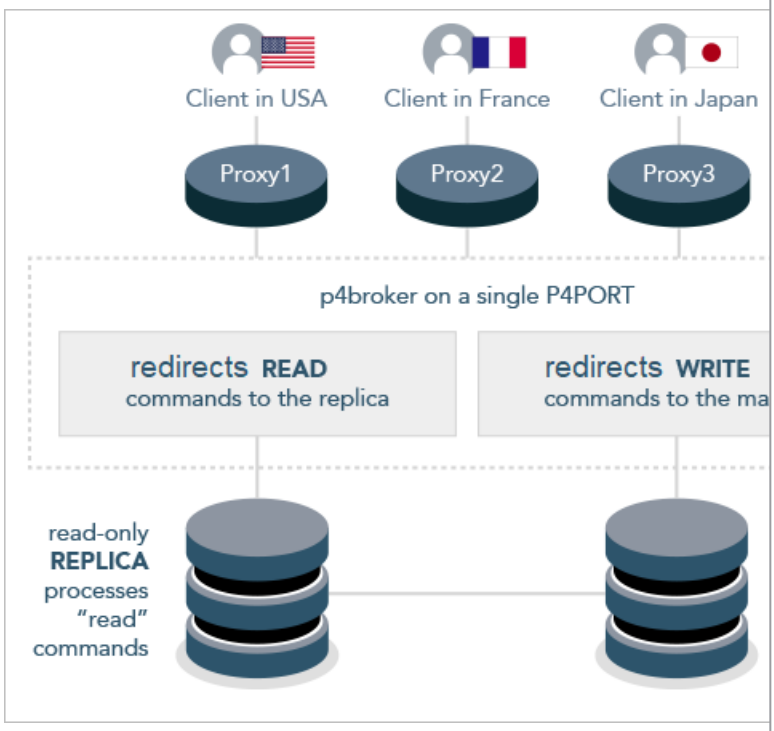
Architecture	Advantage	Disadvantage
Forwarding replica  **Note** A master is a standard server type that doesn't support edge servers.	■ customizable filtering ■ supports "daisy chaining" to additional sites. For example, a site in the Philippines might forward to a site in Taiwan that forwards to a site in Japan that forwards to the Master in France. This alternative to directly connecting each Asian	■ "write" commands are slower because local metadata must be pulled from the master ■ requires machine provisioning and administration. See "Configuring a forwarding replica" on page 57. **Tip**

Architecture	Advantage	Disadvantage
	site to Europe: • reduces the metadata replication load on the master server • reduces the amount	Starting with 2018.2, we recommend a **standby** server with rpl.journalcopy.location=1 for high availability and disaster recovery.

Architecture	Advantage	Disadvantage
	of data traveling across the WAN from Europe to Asia	

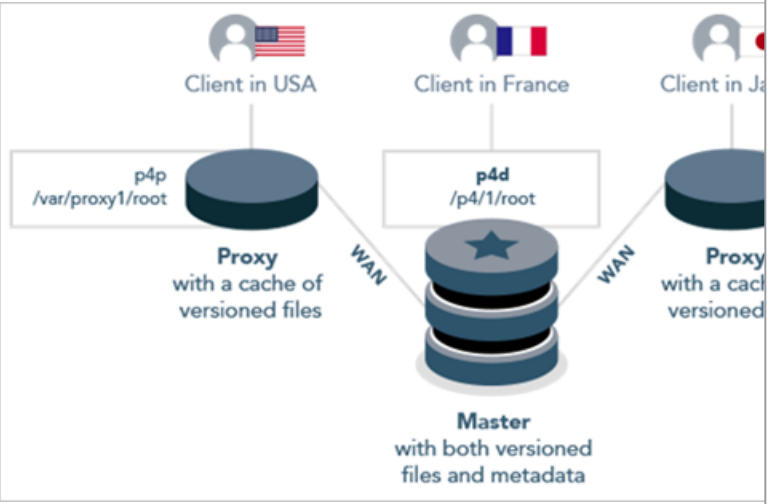
Architecture	Advantage	Disadvantage
	For more information, see the Knowledge Base article on server-to-server arrangements, "Helix replication rules".	

Architecture	Advantage	Disadvantage
"Edge-to-edge chaining" on page 69 	■ 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 101 	■ 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 113) ■ When	■ 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 comm

Architecture	Advantage	Disadvantage
	the p4d process is offline for maintenance, the broker can display a message such as "This server is undergoing maintenance", which is more user-friendly than a TCP connect error. **Tip**	ands". ■ Command "triggers" are an alternative to some traditional broker uses cases. See "Triggering before or after commands".

Architecture	Advantage	Disadvantage
	Note that during the Failover process, such a message is visible to the end-users without using a broker.	

Architecture	Advantage	Disadvantage
"Helix Proxy" on page 121 	■ 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 arti

Architecture	Advantage	Disadvantage
		cle on Pro xy Per for ma nce .

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 48
> - the Support Knowledgebase article, "Replica Types and Use Cases"

Guidelines for setting up multi-site services

This section describes guidelines for setting up a multi-site environment.

General guidelines

Follow these guidelines to simplify the administration and maintenance of your multi-site environments:

- Assign a **server ID** to every server
  - it is best if the serverID is the same as the server name
  - 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 73.

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.

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, which is then valid across all replica instances. 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 the same value for all servers participating in a distributed configuration.
- Enable `rpl.forward.login` (set to `1`) for each replica participating in a distributed configuration.

There might be a slight lag while you wait for each instance to replicate the `db.user` record from the target server.

Connecting services

Services working together in a multi-site 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 Level 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). 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 the "Communicating port information" topic in the *Helix Core Server Administrator Guide: Fundamentals*.

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 that the `P4TICKETS` variable is correctly set when the user (often a script or other automation tool) invokes the downstream service. This enables the downstream service to correctly read the ticket file and authenticate itself as the service user 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" in *Helix Core Server Administrator Guide: Fundamentals*.

Backing up and upgrading services

Backing up and upgrading services in a multi-site environment involve special considerations.

Backing up services

How you back up a service in your multi-site 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 in the *Helix Core Server Administrator Guide: Fundamentals*. If you are using an edge-commit architecture, both the commit server and the edge servers must be backed up. Use the instructions given in "Backup and high availability/disaster recovery (HA/DR) planning" on page 97. ■ 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 the Support Knowedge Base articles on "Offline Checkpoints" and Taking Checkpoints on Edge and Replica Servers, especially the section on "Detecting Coordinated Checkpoint Completion" ■ 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" in Helix Core Server Administrator Guide: Fundamentals. ■ Journal rotation on a replica or edge server begins AFTER the master has completed its journal rotation

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
Upgrading to 2019.1 (and later) is different from previous upgrade. See Upgrading the server > From 2013.3 or later to 2019.1 (distributed environment) in *Helix Core Server Administrator Guide: Fundamentals*.

Configuring centralized authorization and changelist servers

As an alternative to the options at the "Introduction to multi-site deployment architectures" on page 13, it is possible to use a collection of specialized servers that have a shared user base. Two use cases are:

- to simplify user authentication
- to guarantee unique change list numbers across the organization.

The following subsections explain how you create and use these servers: `P4AUTH` for centralized authentication and `P4CHANGE` to generate unique changelist numbers.

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 109.

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 42 and not use the default user named **remote**. See Restricting access to remote depots in Helix Core Server Administrator Guide: Fundamentals.
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 42 and not use the default user named **remote**. See Restricting access to remote depots in Helix Core Server Administrator Guide: Fundamentals.
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.

Command	Forwarded to auth server?	Note
`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.
`p4 logout`	Yes	Command is forwarded to the central server for ticket invalidation.

Limitations and notes

- 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; 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.

Verifying shelved files

The verification of shelved files lets you know whether your shelved archives have been lost or damaged.

If a shelf is local to a specific edge server, you must issue the **`p4 verify -S`** command on the edge server where the shelf was created. If the shelf was promoted, run the **`p4 verify -S`** on the commit server.

You may 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 for shelved archives that are present on the master; that is, for a shelf that was originally created on the master or that was promoted if it was created on an edge server.

Helix server replication

This topic assumes you have read the "Introduction to multi-site deployment architectures" on page 13.

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 "Configuring a forwarding replica" on page 57.)

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 29 and "Helix Broker" on page 101.

Note
Most replica configurations are intended for reading of data. If you require read and write access to a remote server, use a forwarding replica, a distributed Perforce service, or the Helix Proxy. See "Configuring a forwarding replica" on page 57, "Commit-edge" on page 77 and "Helix Proxy" on page 121.

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
- Verifying Replica Integrity
- 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)

System requirements

Replica servers must match the master server in the following:

- release version - see "Upgrading replica servers" on page 56 and the Support Knowledgebase article, "Upgrading Replica Servers"
- 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

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.

Replication basics

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.
`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.)

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 101 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.

The 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`	

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 command creates a small text file named `server.id` in the root directory of the server. The server executable, `p4d`, can also create this `server.id` file:

```
p4d -r $P4ROOT -xD
```

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 instance 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 types of Helix server users: `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.

Important
Read the "Service users" topic in *Helix Core Server Administrator Guide: Fundamentals*.

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`.

Server options to control metadata and depot access

When you start a replica that points to a master server with **P4TARGET**, you must specify both the **-M** (metadata access) and a **-D** (depot access) options, or 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.

P4TARGET

Set P4TARGET to the 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, specify it on the **p4d** command line with the **-t *protocol*:*host*:*port*** option, or you can use p4 configure to set a **P4TARGET** for each named replica server. See the table below for the available ***protocol*** options.

If you specify a target, **p4d** examines its configuration for startup.*N* commands: if no valid p4 pull commands are found, **p4d** runs and 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. See "p4 pull vs. p4 replicate" on the next page for details.

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 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.

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. 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.

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.

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 105. 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.

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 the proxy 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/...`

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".

Configuring a read-only replica

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 topic in *Helix Core Server Administrator Guide: Fundamentals*.

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 `11111`, 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 `22222`, 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.

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" in *Helix Core Server Administrator Guide: Fundamentals*.
- 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 43.
4. Set the service group protections to **super** in your protections table. See "Permissions for service users" on page 43.
5. Set the level of security to **3** or higher on the master server. See Server security levels in *Helix Core Server Administrator Guide: Fundamentals*.
   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 and "checkpointing options" in *Helix Core Server Administrator Guide: Fundamentals*.
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.

   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 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 "Spec configuration" on page 51:
   ```
   $ 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
    ```

Using the replica

You can perform all normal operations against your master server (**`p4 -p master:11111`** ***`command`***). To reduce the load on the master server, direct reporting (read-only) commands to the replica (**`p4 -p replica:22222`** ***`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:22222 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:22222 client replica_client
Replica does not support this command.
```

However, **p4 client -o** works:

```
$ p4 -p replica:22222 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 Support Knowledge base article, "Using P4Broker to redirect read-only commands".

See also the chapter on "Helix Broker" on page 101.

Next step

"Upgrading replica servers" on the facing page

Upgrading replica servers

Upgrade replicas before upgrading the master

We recommend that you first upgrade any server instance replicating 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 that server:

```
p4d -M readonly -D readonly -In Replica2 -p 6661 -r
/usr/replica/root -J journal -d
```

Configuring a 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 the previous page.

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" in *Helix Core Server Administrator Guide: Fundamentals*.
- 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 43.
4. Set the service user group protections to `super` in your protections table. See "Permissions for service users" on page 43.
5. Set the level of security to `3` or higher on the master server. See Server security levels in *Helix Core Server Administrator Guide: Fundamentals*.
   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" on the facing page

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 and "checkpointing options" in *Helix Core Server Administrator Guide: Fundamentals*.
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 58:
   ```
   $ 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
    ```

Configuring a build server (also known as build farm server)

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.

With a buiild server, you can offload the workload of the automated build processes onto a separate machine. This ensures that your main Helix server's resources are available to your users for their normal day-to-day tasks.

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. As discussed in "Commit-edge" on page 77, 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.

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 56.

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" in *Helix Core Server Administrator Guide: Fundamentals*.
- 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 "Tickets and timeouts for service users" on page 43.
4. Set the service user group protections to `super` in your protections table. See "Permissions for service users" on page 43.
5. Set the level of security to `3` or higher on the master server. See Server security levels in *Helix Core Server Administrator Guide: Fundamentals*.
   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 and "checkpointing options" in *Helix Core Server Administrator Guide: Fundamentals*.
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 specified in the "Spec configuration" on page 58:

   ```
   $ 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.

Configuring a replica with shared archives

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.

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.

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.

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 57.

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 56.

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 a distant commit server. You can have any number of such edge servers in the chain. Among the many possible scenarios are:

1. **`edge3 -> edge2 -> edge1 -> commit server`**
2. **`edge3 -> edge2 -> edge1 -> forwarding replica -> commit`**
3. **`edge3 -> edge2 -> edge1 -> forwarding replica 2 -> forwarding replica 1 -> commit`**

If you include a forwarding replica, it must connect directly with another forwarding replica or the commit server. Do NOT put an edge server between a forwarding replica and the commit server

Configure each server so that its P4TARGET is the closest inner server. For example, **`edge2`**'s target is **`edge1`**.

Configure each server with a service user with ticket-based authentication to ALL of the servers that are closer to the commit server. For example, in scenario 1 above, **`edge2`** needs to be authenticated with both **`edge1`** and the **`commit server`**.

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

Also, in the case of edge-to-edge chaining (version 2019.1 and later), 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.

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:

E x a m p l e 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/...
```

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.

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.

Verifying replica integrity

Tools to ensure data 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" in *Helix Core Server Administrator Guide: Fundamentals*.

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

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 desired levels of integrity checking:

- rpl.checksum.auto
- rpl.checksum.change
- rpl.checksum.table

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.)

Valid settings for **`rpl.checksum.auto`** are:

`rpl.checksum.auto`	Database tables checked with every journal rotation
`0`	No checksums are performed.
`1`	Verify only the most important system and revision tables: `db.archmap, db.config, db.depot, db.graphindex, db.graphperm, db.group, db.groupx, db.integed, db.integtx, db.ldap, db.object, db.protect, db.pubkey, db.ref, db.rev, db.revcx, db.revdx, db.revhx, db.revtx, db.stream, db.submodule, db.ticket, db.trigger, db.user`
`2`	Verify all database tables from level 1, plus: `db.bodtext, db.bodtextcx, db.bodtexthx, db.counters, db.excl, db.fix, db.fixrev, db.haveview, db.ixtext, db.ixtexthx, db.job, db.logger, db.message, db.nameval, db.property, db.remote, db.repo, db.revbx, db.review, db.revsx, db.revux, db.rmtview, db.server, db.svrview, db.traits, db.uxtext`
`3`	Verify all metadata, including metadata that is likely to differ, especially when comparing an upstream server with a build-farm or edge-server replica.

Valid settings for **`rpl.checksum.change`** are:

`rpl.checksum.change`	Verification performed with each changelist
`0`	Perform no verification.

rpl.checksum.change	Verification performed with each changelist
1	Write a journal note when a p4 submit, p4 fetch, p4 populate, p4 push, or p4 unzip command completes. The value of the rpl.checksum.change configurable will determine the level of verification performed for the command.
2	Replica verifies changelist summary, and writes to integrity.csv if the changelist does not match.
3	Replica verifies changelist summary, and writes to integrity log even when the changelist does match.

Valid settings for **rpl.checksum.table** are:

rpl.checksum.table	Level of table verification performed
0	Table-level checksumming only.
1	When a table is unloaded or scanned, journal notes are written. These notes are processed by the replica and are logged to integrity.csv if the check fails.
2	When a table is unloaded or scanned, journal notes are written, and the results of journal note processing are logged even if the results match.

For more information, see **p4 help journaldbchecksums**.

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`

Commit-edge

This topic assumes you have read the "Introduction to multi-site deployment architectures" on page 13.

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 "Helix server replication" on page 34, including the section on "Connecting services" on page 25, which includes information on "Managing SSL key pairs" on page 26.

Tip
An edge server can be used instead of a build server, and this usage is referred to as a *build edge 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 87.

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 service user accounts for the commit and edge servers:

   ```
   $ p4 user -f svc_chicago_commit
   $ p4 user -f svc_tokyo_edge
   ```

   and in the user spec, set the user `Type:` field to `service`.

2. To prevent the service user logins from timing out, add the service users to a group with an unlimited timeout:

   ```
   $ p4 group no_timeout
   ```

   and in the `group` spec, set the `Users:` field to include the `svc_chicago_commit` and `svc_tokyo_edge` service users, and set the `Timeout:` field to `unlimited`.

3. Assign passwords to the service user accounts by providing a value at the prompts.

   ```
   $ p4 passwd svc_chicago_commit
   $ p4 passwd svc_tokyo_edge
   ```

4. 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. 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. Set the `server ID` of the commit server:

   ```
   $ p4 serverid chicago_commit
   ```

3. Create the edge server specification:

   ```
   $ p4 server -c edge-server tokyo_edge
   ```

   and 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
   "db.have,db.working,db.locks,db.resolve,db.revsh,db.workingx,
   db.resolvex,db.stash,db.haveg,db.workingg,db.locksg,db.resolv
   eg" -z -jd edge.ckp
   ```

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
   chicago.perforce.com:1666 trust
   ```

   before creating the service user login ticket:

   ```
   $ p4 -E P4TICKETS=/tokyo/p4root/.p4tickets -u svc_tokyo_edge
   -p 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 UNIX and Windows in the "Installing and Upgrading the Server" chapter of the *Helix Core Server Administrator Guide: Fundamentals*.

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/.p4tickets -u svc_chicago_
commit -p tokyo.perforce.com:1666 trust
```

before creating the service user login ticket:

```
$ p4 -E P4TICKETS=/chicago/p4root/.p4tickets -u svc_chicago_
commit -p 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 25
> - 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
```

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.

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`).

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" on the next page explains what sort of existing replicates can be profitably replaced with edge servers.
- "Deploying commit and edge servers incrementally" on the next page describes an incremental approach to migration.
- "Hardware, sizing, and capacity" on the next page discusses how provisioning needs shift as you migrate to the edge-commit architecture.
- "Migration scenarios" on page 89 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. See "Migrating a workspace from a commit server or remote edge server to the local edge server" on page 91 for details.

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, email support@perforce.com.

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. Take a checkpoint of your central server, filtering out the **`db.have`**, **`db.working`**, **`db.resolve`**, **`db.locks`**, **`db.revsh`**, **`db.workingx`**, **`db.resolvex`** tables.

     ```
     $ p4d -K
     "db.have,db.working,db.resolve,db.locks,db.revsh,db.w
     orkingx,db.resolvex"
           -jd my_filtered_checkpoint_file
     ```

     > **Tip**
     > If you want to produce a filtered journal dump file, go to *Helix Core Server Administrator Guide: Fundamentals*, and look in the "Helix Core server Reference" for the **`-k`** and **`-K`** options.

     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 **`db.have`**, **`db.working`**, **`db.resolve`**, **`db.locks`**, **`db.revsh`**, **`db.workingx`**, **`db.resolvex`** tables.
7. Start the replica; it 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 it seems very attractive to be able to continue to use those clients after the conversion from a build-server to an edge server. There is one small detail:

- 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 can be 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:

```
$ rm db.have db.view db.locks db.working db.resolve db.revsh
db.workingx db.resolvex
$ mv db.have.rp db.have
$ mv db.view.rp db.view
```

4. Start the server; it is now an edge server and all of its locally-bound clients can continue to 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, please contact Technical Support.

Migrating a workspace from a commit server or remote edge server to the local edge server

Scenario: You have a workspace on a commit or remote edge server that you want to move to the local edge server.

1. Current work may be unsubmitted and/or shelved.
2. Execute the following command against the local edge server, where the workspace is being migrated *to*. ***protocol*:*host*:*port*** refers to the commit or remote edge server the workspace is being migrated *from*.

```
$ p4 reload -c workspace -p protocol:host:port
```

Managing distributed 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 high availability/disaster recovery (HA/DR) planning" on page 97 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 facing 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. For more information, see "Automatically promoting shelves" below.
- You must explicitly promote a shelf when the commit and edge servers do not share the archive. For more information, see "Explicitly promoting shelves" on the next page.

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 page 95 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 fields 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

This section explains how you manage existing triggers 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. For more information on triggers, see the *Helix Core Server Administrator Guide: Fundamentals*.

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%"
```

Backup and high availability/disaster recovery (HA/DR) planning

A commit server can use the same backup and 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.

Helix Broker

This topic assumes you have read the "Introduction to multi-site deployment architectures" on page 13.

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.

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 Linux package-based installation Prerequisites in *Helix Core Server Administrator Guide: Fundamentals*.

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:
   ```
   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)

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/

See "Configuring the broker" on page 109.

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/

See "Configuring the broker" on page 109.

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/

See "Configuring the broker" on page 109.

Running the broker

After you have created your configuration file (see "Configuring the broker" on page 109), 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. See "Enabling SSL support" on page 46. 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** (typically a service user) 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.

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 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.

Option	Meaning
`-q`	Run quietly (no startup messages).
`-V`	Print broker version, and then exit.
`-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 (e.g. `127.0.0.1`) is used, the transport is always `tcp4`, and if an IPv6 literal address (e.g. `::1`) is used, then the transport is always `tcp6`.
`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 more than one value; see example.	`debug-level = server=1;` `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 login -u`** ***`service-user`*** **`-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 119.
	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 (e.g. `[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
`clientVersion:`	Version of client application program

Command detail	Definition
`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 115 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 "Helix server replication" on page 34. The broker works with both metadata-only replicas and with replicas that have access to both metadata and versioned files.

There is no limit to the number of alternate replica servers you can define 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 assigned to the alternate server is used in command handler specifications. See "Command handler specifications" on page 113.

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`**) and Helix servers at version 2007.2 or later. For more information about setting up a central authorization server, see "Configuring centralized authorization and changelist servers" on page 29.

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).

"Defending from man-in-the-middle attacks" on page 127

Helix Proxy

This topic assumes you have read the "Introduction to multi-site deployment architectures" on page 13.

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:

- "Enabling SSL support" on page 46
- "Defending from man-in-the-middle attacks" on page 127

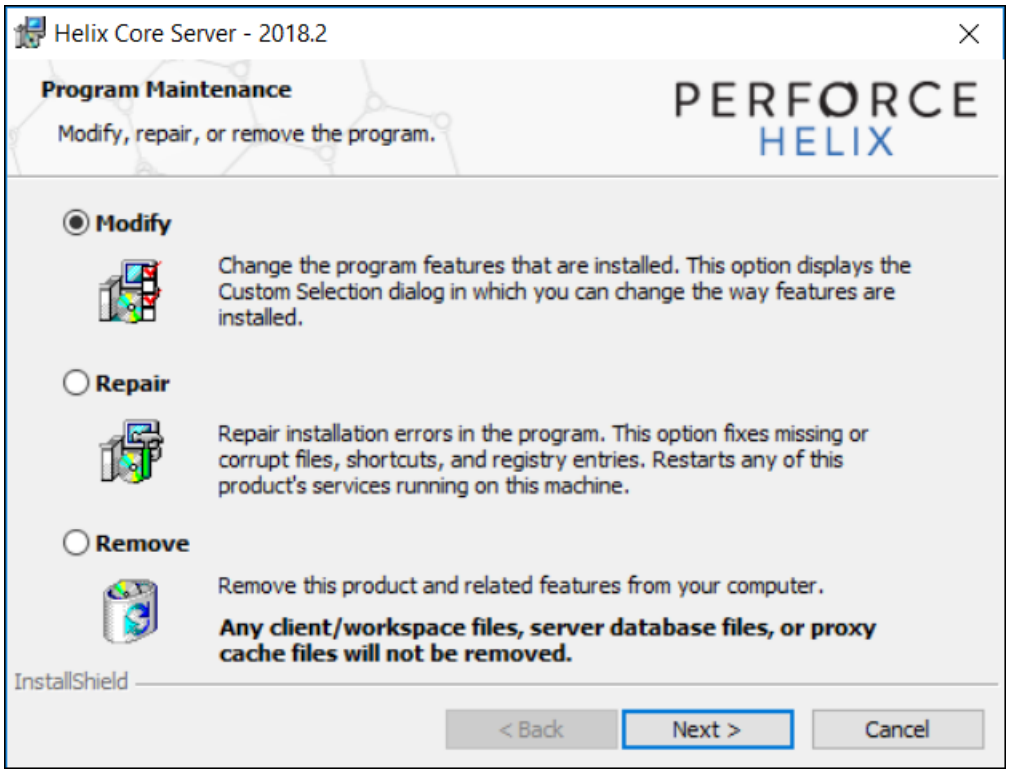
UNIX

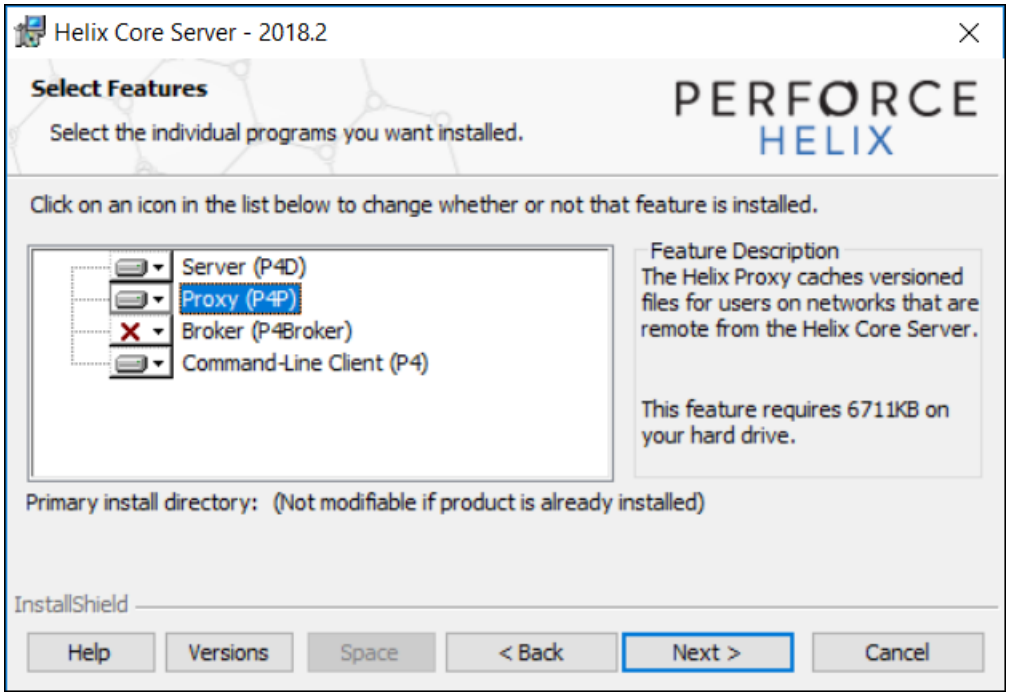
To install P4P on UNIX or Linux, do the following:

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.
PERFORCE
HELIX
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.
InstallShield
< Back
Next >
Cancel


Helix Core Server - 2018.2
Select Features
Select the individual programs you want installed.
PERFORCE
HELIX
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)
InstallShield
Help
Versions
Space
< Back
Next >
Cancel

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 the *Helix Core Server Administrator Guide: Fundamentals* for more information.

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).

Option	Meaning
`-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.
`-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.

Option	Meaning
`-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 instances of `p4p`. 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.
-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

Option	Meaning
`-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 "Enabling SSL support" on page 46. 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" in *Helix Core Server Administrator Guide: Fundamentals*), 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.
>
> You do not need to stop the proxy to delete its cached files or the `pdb.lbr` file.

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" in *Helix Core Server Administrator Guide: Fundamentals*.

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

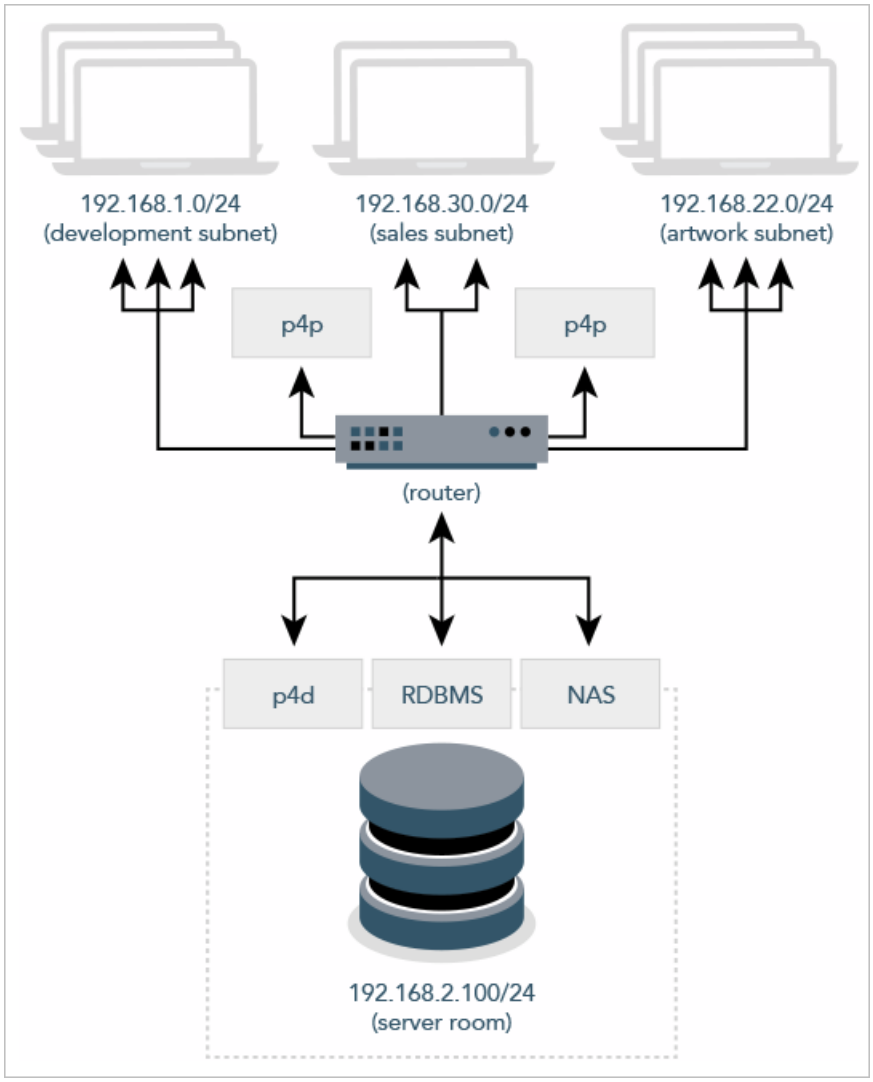
In addition to the topics in this chapter, see the Support Knowledgebase article on tuning tips for "Proxy Performance", including how to minimize the syncing of small files.

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`**.

Helix Core server (p4d) Reference

Start the Perforce service or perform checkpoint/journaling (system administration) tasks.

Syntax

```
p4d [ options ]
p4d.exe [ options ]
p4s.exe [ options ]
p4d -j? [ -z | -Z ] [ args ... ]
```

Description

The first three forms of the command invoke the background process that manages the Helix server versioning service.

The fourth form is for system administration tasks involving 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 page 136, "Checkpointing options" on page 136, "Journal restore options" on page 139, "Replication and multi-server options" on page 140, "Journal dump and restore filtering" on page 141, "Certificate handling" on page 142, and "Configuration options" on page 142.

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).

Server options	Meaning
`--pid-file[=file]`	Write the PID of the server to a file named `server.pid` in the directory specified by `P4ROOT`, or write the PID to the file specified by *`file`*. This makes it easier to identify a server instance among many. The *`file`* parameter can be a complete path specification. The file does not have to reside in `P4ROOT`.
`--daemonsafe`	Is like `-d` and forks the `p4d` into the background, but also closes the stdio (standard input output) files.

Server options	Meaning
`-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*.
`-xu`	Run database upgrades and exit. 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.
`-xvU`	Run fast verification. Do not lock database tables, and verify only that the unlock count for each table is zero.
`-xD [`*`serverID`*`]`	Display (or set) the server's *`serverID`* (stored in the `server.id` file) and exit.

General options

General options	Meaning
`-h`, `-?`	Print help message.
`-V`	Print version number.
`-A` *`auditlog`*	Specify an audit log file. Overrides `P4AUDIT` setting. Default is null.
`-Id` *`description`*	A server description for use with `p4 server`. Overrides `P4DESCRIPTION` setting.
`-In` *`name`*	A server name for use with `p4 configure`. Overrides `P4NAME` setting.
`-J` *`journal`*	Specify a journal file. Overrides `P4JOURNAL` setting. Default is `journal`. (Use `-J off` to disable journaling.)
`-L` *`log`*	Specify a log file. Overrides `P4LOG` setting. Default is `STDERR`.
`-p` *`port`*	Specify a port to listen to. Overrides `P4PORT`. Default `1666`.
`-r` *`root`*	Specify the server root directory. Overrides `P4ROOT`. Default is current working directory.
`-v` *`subsystem=level`*	Set trace options. Overrides value `P4DEBUG` setting. Default is null.
`-C1`	Force the service to operate in case-insensitive mode on a normally case-sensitive platform.
`--pid-file [=`*`name`*`]`	Write the server's PID to the specified file. Default name for the file is `server.pid`.

Checkpointing options

Checkpointing options	Meaning
`-c` *`command`*	Lock database tables, run *`command`*, unlock the tables, and exit.

Checkpointing options	Meaning
`-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.
`-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

- 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`

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, in conjunction with the source revision, used to help determine what integration changes should be applied to the target revision. 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. The default depot name is depot. See also default depot and stream 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: Fundamentals" 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.

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.

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.

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 that you are changing in your client workspace that is checked out. If the file is not checked out, opening it in the file system does not mean anything to the versioning engineer.

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 additional intelligence 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.

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: Fundamentals" 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.

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.