HelixCore

Helix Versioning Engine Administrator Guide: Multi-Site Deployment

2017.1

May 2017

PERFORCE

www.perforce.com

Contents

How to Use this Guide

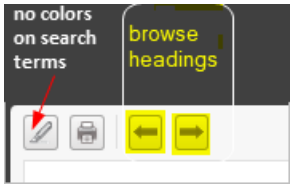
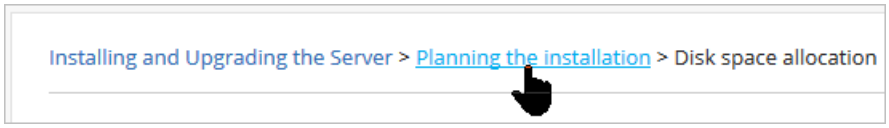
This manual is intended for administrators responsible for installing, configuring, and maintaining multiple interconnected or replicated Perforce services. Administrators of sites that require only one instance of the Perforce service will likely find the *Helix Versioning Engine Administrator Guide: Fundamentals* sufficient.

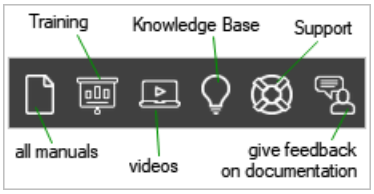
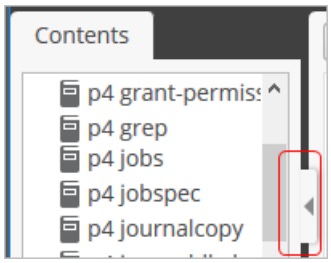
This guide assumes familiarity with the material in the *Helix Versioning Engine Administrator Guide: Fundamentals*.

Search within this guide

Use quotes for an exact multi-word phrase:	"revision range" Revision ranges are also acceptable.
Quickly spot multiple search terms in color-coded results (different color for each term):	branch tag repo Force an overwrite to the branch. Delete the repository's branch or tag
Find search terms on page with **Command-F** on Mac or **CTRL+F** on Windows	

Navigation

Browse to the next or previous heading with arrow buttons:	
See the top of any page to knows its location within the book:	

Use the links to resources at the footer of each page:	
Resize the Content pane as needed:	

Tip
When sharing URLs, you can ignore the extra characters at the end of each page's URL because standard URLs do work. For example:

```
https://www.perforce.com/perforce/doc.current/manuals/cmdref/#CmdRef/p4_add.htm
```

or

```
https://www.perforce.com/perforce/doc.current/manuals/cmdref/#CmdRef/configurables.configurables.html#auth.default.method
```

Feedback

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

Other Helix Core documentation

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

Notation	Meaning
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.
...	■ Repeats as much as needed: • `alias-name[[$(arg1)... [$(argn)]]=transformation` ■ Recursive for all directory levels: • `clone perforce:1666 //depot/main/p4... ~/local-repos/main` • `p4 repos -e //gra.../rep...`
element1 | *element2*	Either *element1* or *element2* is required.

What's new in this guide

This section provides a list of changes to this guide for the latest release. For a list of all new functionality and major bug fixes in the latest Helix Versioning Engine release, see http://www.perforce.com/perforce/doc.current/user/relnotes.txt.

Updates and corrections

Make `p4 lock` on an edge server take global locks on the commit server by default:: See "Locking and unlocking files" on page 71.

Introduction to Federated Services

Perforce federated architecture aims to make simple tasks easy and complex tasks possible; it allows you to start simply and to grow incrementally in response to the evolving needs of your business.

This chapter describes the different types of Perforce servers and explains how you combine these to solve usability and performance issues. In addition, this chapter discusses the issues that affect service federation independently of the particular type of service used, issues like user authentication and communication among federated services. Subsequent chapters in this book describe each type of service in detail.

To make best use of the material presented in this book, you should be familiar with the *Helix Versioning Engine Administrator Guide: Fundamentals*.

Overview

Helix Versioning Engine Administrator Guide: Fundamentals explains how you create, configure, and maintain a single Helix Versioning Engine. For most situations, a single server that is accessible to all users can take care of their needs with no problems. However, as business grows and usage expands, you might find yourself needing to deploy a more powerful server-side infrastructure. You can do so using three different types of Perforce services:

- ***Proxy***

  Where bandwidth to remote sites is limited, you can use a Perforce proxy to improve performance by mediating between Perforce clients and the versioning service. Proxies cache frequently transmitted file revisions. By intercepting requests for cached revisions, the proxy reduces demand on the server and keeps network traffic to a minimum.

  The work needed to install and configure a proxy is minimal: the administrator needs to configure a proxy on the side of the network close to the users, configure the users to access the service through the proxy, and then configure the proxy to access the Helix Core versioning service. You do not need to backup the proxy cache directory: in case of failure, the proxy can reconstruct the cache based on the Helix Core server metadata. For complete information about using proxies, see "Helix Proxy" on page 96.

- ***Broker***

  A Perforce broker mediates between clients and server processes (including proxies) to implement policies in your federated environment. Such policies might direct specific commands to specific servers or they might restrict the commands that can be executed. You can use a broker to solve load-balancing, security, or other issues that can be resolved by sorting requests directed to one or more Helix Core servers.

  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 Core 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 backup the broker. In case of failure, you just need to restart it and make sure that its configuration file has not been corrupted. For complete information about using the broker, see "The Helix Broker" on page 77.

- ***Replica***

  A replica duplicates server data; it is one of the most versatile elements in a federated architecture. You can use it to provide a warm standby server, to reduce load and downtime on a primary server, to support build farms, or to forward requests to a central server. This latter use case, of the forwarding replica, can be implemented using a commit-edge architecture, which improves performance and addresses the problem of remote access.

  The amount of administrative work needed for installing, configuring, and managing replicates varies with the type of replicate used. For information about the handling of different replicate types, see "Perforce Replication" on page 21. For information about commit-edge deployments, see "Commit-edge Architecture" on page 56.

In addition to these three types of servers, to simplify administrative work, a federated architecture might also include servers dedicated to centralized authorization and changelist numbering. For more information, see "Configuring centralized authorization and changelist servers" on page 17. The next section explains how you might combine these types to address various user needs.

User scenarios

Which types of servers you use and how you combine them varies with your needs. The following discussion examines what servers you'd choose to support high availability, geographical distribution, build automation, scalability, and governance.

Availability

As users become more dependent on a Helix Core server, you might want to minimize server downtime. By deploying additional replicas and brokers, you can set up online checkpoints so that users can continue work while you make regular backups. This more sophisticated infrastructure also allows you to perform regular maintenance using `p4 verify` or `p4 dbverify` without server downtime. You can re-route requests targeted for one machine to another during routine system maintenance or operating system upgrades.

Should your primary server fail, this server infrastructure allows you to fail over to a standby machine with minimal downtime. If the backup server is deployed to a different machine, this architecture can also be used for disaster recovery. Replica types best suited for failover and disaster recovery are read-only replicas and forwarding replicas.

Remote offices

As your organization grows and users are added in remote offices, you need to provide remote users with optimal performance when they access the Helix Core server.

The primary challenge to performance in a geographically distributed environment is network latency and bandwidth limitations. You can address both issues using Perforce proxies and replicas. These reduce network traffic by caching file content and metadata in the remote office and by servicing requests locally whenever possible. Up to 98% of user requests can be handled by the servers in the remote office.

You can also configure brokers to re-route requests in cases of outage and to assist in managing off-hour requests that occur with a global workforce.

Build/test automation

With the increasing use of build and test automation, it is important to deploy a server architecture that addresses the growing workload. Automated build and test tools can impose a massive workload on the storage and networking subsystems of your Helix Core server. This workload includes agile processes and continuous delivery workflows, reporting tools that run frequent complex queries, project management tools that need close integration with server data, code search tools, static analysis tools, and release engineering tools that perform automated branch integrations and merges.

To improve user experience, you need to shift this growing workload to dedicated replica servers and relieve your master server of those tasks, enabling it to concentrate on servicing interactive user requests.

Scalability

As your organization grows, you need to grow your infrastructure to meet its needs.

- The use of advanced software engineering tools will benefit from having additional server-side resources. Deploying Perforce proxies, replicas, and brokers allows you to add additional hardware resources and enables you to target each class of request to an appropriately-provisioned server, using your highest-performance infrastructure for your most critical workloads while redirecting lower-priority work to spare or under-utilized machines.
- As the number of users and offices grows you can plan ahead by provisioning additional equipment. You can deploy Perforce proxies, replicas, and brokers to spread your workload across multiple machines, offload network and storage usage from your primary data center to alternate data centers, and support automation workloads by being able to add capacity.

Policy-based governance

As usage grows in size and sophistication, you might want to establish and maintain policies and procedures that lead to good governance.

For example, you might want to use repository naming conventions to make sure that your repository remains well organized and easy to navigate. In addition you might find that custom workflows, such as a change review process or a release delivery process, are best supported by close integration with your version control infrastructure.

You can use brokers in your federated deployment to filter requests, enforce policy, and re-route commands to alternate destinations. This provides a powerful infrastructure for enforcing your organization's policies. Deploying trigger scripts in your servers instances enables additional integration with other software development and management tools.

Setting up federated services

This section describes some of the issues that administration must address in setting up a federated environment.

General guidelines

Following these guidelines will simplify the administration and maintenance of federated environments:

- Every server should be assigned a server ID; 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.
- Every server should have an assigned (and preferably unique) service user name. This simplifies the reading of logs and provides authentication and audit trails for inter-server communication. Assign service users strong passwords. Use the `p4 server` command to assign a service user name.
- Enable structured logging on all your services. Doing so can greatly simplify debugging and analysis, and is also required in order to use the `p4 journaldbchecksums` command to verify the integrity of a replica.
- Configure each server to reject operations that reduce its disk space below the limits defined by that service's `filesys.*.min` configurables.
- 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 52 for details.

Authenticating users

Users must have a ticket for each server they access in a federated environment. 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 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 federated 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 (and mandatory at security level 4) 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.

Managing trust between services

The user that owns the server, broker, or proxy process is typically a service user. As the administrator, you must create a `P4TRUST` file on behalf of the service user by using the `p4 trust` command) that recognizes the fingerprint of the upstream Perforce service.

By default, a user's `P4TRUST` file resides in their home directory as `.p4trust`. Ensure that the `P4TRUST` variable is correctly set so that when the user (often a script or other automation tool) that actually invokes the `p4d`, `p4p`, or `p4broker` executable is able to read filename to which `P4TRUST` points in the invoking user's environment.

Further information is available in the *Helix Versioning Engine 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. Follow these steps 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 `P4TICKET` variable is correctly set when the user (often a script or other automation tool) that actually invokes the downstream service, does so, so that the downstream service can 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.

The process for creating a key pair is the same as it is for any other server: set `P4SSLDIR` to a valid directory with valid permissions, and use the following commands to generate pairs of `privatekey.txt` and `certificate.txt` files, and make a record of the key's fingerprint.

- Server: use `p4d -Gc` to create the key/certificate pair and `p4d -Gf` to display its fingerprint.
- Broker: use `p4broker -Gc` to create the key/certificate pair and `p4broker -Gf` to display its fingerprint.
- 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. Further information is available in the *Helix Versioning Engine Administrator Guide: Fundamentals*.

Backing up and upgrading services

Backing up and upgrading services in a federated environment involve special considerations. This section describes the issues that you must resolve in this environment.

Backing up services

How you backup federated services depends upon the service type:

- **Server**

  Follow the backup procedures described in the *Helix Versioning Engine 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 74.

  Backup requirements for replicas that are not edge servers vary depending on your site's requirements.

- Broker: the broker stores no data locally; you must backup its configuration file manually.
- Proxy: the proxy requires no backups and, if files are missing, automatically rebuilds its cache of data. The proxy contains no logic to detect when diskspace is running low. Periodically monitor your proxy to ensure it has sufficient diskspace for continued operation.

Upgrading services

Servers, brokers, and proxies must be at the same release level in a federated environment. When upgrading use a process like the following:

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.

Configuring centralized authorization and changelist servers

There are cases where rather than using federated services you want to use a collection of servers that have a shared user base. In this situation, you probably want to use specialized servers to simplify user authentication and 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 Perforce 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. If a user exists on both the central authorization server and the outer server, the most permissive protections of the two lines of the protections table are assigned to the user.

You can use any existing Helix Versioning Engine 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 Versioning Engine 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 83.

In the following example, an outer server (named `server2`) 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 -In server2 -a central:1666 -p 1999
```

Note
On Windows, configure the outer server with `p4 set -S` as follows:

```
C:\> p4 set -S "Outer Server" P4NAME=server2
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?	Notes
`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	Password settings are stored on, and must meet the security level requirements of, the central server.
`p4 review`	No	Service user (or `remote`) must have access to the central server.
`p4 reviews`	No	Service user (or `remote`) must have access to the central server.
`p4 user`	Yes	Local user data is derived from the central server.
`p4 users`	Yes	Local user data is derived from the central server.
`p4 protect`	No	The local server's protections table is displayed if the user is authorized (as defined by the combined protection tables) to edit it.
`p4 protects`	Yes	Protections are derived from the central server's protection table as appended to the outer server's protection table.
`p4 login`	Yes	Command is forwarded to the central server for ticket generation.

Command	Forwarded to auth server?	Notes
`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, Perforce servers do not coordinate the numbering of changelists. Each Helix Versioning Engine 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 a Helix Versioning Engine 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 -In server2 -g central:1666 -p 1999
```

> **Note**
> On Windows, configure the outer server with `p4 set -S` as follows:
>
> ```
> C:\> p4 set -S "Outer Server" P4NAME=server2
> C:\> p4 set -S "Outer Server" P4CHANGE=central:1666
> C:\> p4 set -S "Outer Server" P4PORT=1999
> ```

In this example, the outer server (named `server2`) 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.

Perforce Replication

Replication is the duplication of server data from one Helix Versioning Engine to another Helix Versioning Engine, 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 and reports, builds, and checkpoints can be run against a replica server, reducing lock contention. For checkpoints and some reporting tasks, only metadata needs to be replicated. For reporting and builds, replica servers need access to both metadata and versioned 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.

Combined with a centralized authorization server (see "Centralized authorization server (P4AUTH)" on page 17), Perforce administrators can configure the Helix Broker (see "The Helix Broker" on page 77) to redirect commands to replica servers to balance load efficiently across an arbitrary number of replica servers.

> **Note**
> Most replica configurations are intended for reading of data. If you require read/write access to a remote server, use either a forwarding replica, a distributed Perforce service, or the Helix Proxy. See "Configuring a forwarding replica" on page 43, "Commit-edge Architecture" on page 56 and "Helix Proxy" on page 96 for details.

System requirements

- As a general rule, *All replica servers must be at the same release level or at a release later as the master server.* Any functionality that requires an upgrade for the master requires an upgrade for the replica, and vice versa.
- All replica servers must have the same Unicode setting as the master server.

- All replica servers must be hosted on a filesystem with the same case-sensitivity behavior as the master server's filesystem.
- `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 may be active at one time.
- The replica server does not need a duplicate license file.
- The master and replica servers must have the same time zone setting.

> **Note**
> 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.

Replication basics

Replication of Helix Core servers depends upon several commands 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. In order 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 Versioning Engine. 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.

Command or Feature	Typical use case
`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.)

Server names
`P4NAME`
`p4d -In name`
	Helix Core 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.

Service users `p4d -u svcuser`	A new type of user intended for authentication of server-to-server communications. Service users have extremely limited access to the depot and do not consume Perforce 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 Core Servers.

Metadata access
`p4d -M readonly`
`db.replication`
	Replica servers can be configured to automatically reject user commands that attempt to modify metadata (`db.*` files).
	In `-M readonly` mode, the Helix Versioning Engine 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.

Command or Feature	Typical use case
Metadata filtering	Replica servers can be configured to filter in (or out) data on client workspaces and file revisions. You can use the `-P serverId` option with the `p4d` command to create a filtered checkpoint based on a serverId. You can use the `-T tableexcludelist` option with `p4 pull` to explicitly filter out updates to entire database tables. Using the `ClientDataFilter:`, `RevisionDataFilter:`, and `ArchiveDataFilter:` fields of the `p4 server` form can provide you with far more fine-grained control over what data is replicated. Use the `-P serverid` option with `p4 pull`, and specify the `Name:` of the server whose `p4 server` spec holds the desired set of filter patterns.

Command or Feature	Typical use case

Depot file access
`p4d -D readonly`
`p4d -D shared`
`p4d -D ondemand`
`p4d -D cache`
`p4d -D none`
`lbr.replication`
	Replica servers can be configured to automatically reject user commands that attempt to modify archived depot files (the "library").
	■ In `-D readonly` mode, the Helix Versioning Engine 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.
	■ In `-D ondemand` mode, or `-D shared` mode (the two are synonymous) the Helix Core 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 `-D cache` mode, the Helix Versioning Engine 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 `-D none` mode, the Helix Versioning Engine 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.
	These options can also be set using `lbr.replication.*` configurables, described in the "Configurables" appendix of the *P4 Command Reference*.

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` and `-D` options, or their equivalent `db.replication` and `lbr.replication` configurables, correctly specified.
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 should be empty or missing, the replica server will refetch 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 "The Helix Broker" on page 77 for details.

Warning
Replication requires uncompressed journals. Starting the master using the `p4d -jc -z` command breaks replication; use the `-Z` flag instead to prevent journals from being compressed.

The p4 pull command

Perforce's `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)]
> ```

Server names and P4NAME

To set a Helix Core server name, set the `P4NAME` environment variable or specify the `-In` command line option to `p4d` when you start the server. Assigning names to servers is essential for configuring replication. Assigning server names permits most of the server configuration data to be stored in Perforce itself, as an alternative to using startup options or environment values to specify configuration details. In replicated environments, named servers are a necessity, because `p4 configure` settings are replicated from the master server along with other Perforce metadata.

For example, if you start your master server as follows:

```
$ p4d -r /p4/master -In master -p master:11111
```

And your replica server as follows:

```
$ p4d -r /p4/replica -In Replica1 -p replica:22222
```

You can use `p4 configure` on the master to control settings on *both* the master and the replica, because configuration settings are part of a Helix Core server's metadata and are replicated accordingly.

For example, if you issue following commands on the master server:

```
$ p4 -p master:11111 configure set master#monitor=2
$ p4 -p master:11111 configure set Replica1#monitor=1
```

After the configuration data has been replicated, the two servers have different server monitoring levels. That is, if you run `p4 monitor show` against `master:11111`, you see both active and idle processes, because for the server named `master`, the `monitor` configurable is set to `2`. If you run `p4 monitor show` against `replica:22222`, only active processes are shown, because for the `Replica1` server, `monitor` is `1`.

Because the master (and each replica) is likely to have its own journal and checkpoint, it is good practice to use the `journalPrefix` configurable (for each named server) to ensure that their prefixes are unique:

```
$ p4 configure set master#journalPrefix=/master_checkpoints/master
$ p4 configure set Replica1#journalPrefix=/replica_
checkpoints/replica
```

For more information, see:

http://answers.perforce.com/articles/KB_Article/Master-and-Replica-Journal-Setup

Server IDs: the p4 server and p4 serverid commands

You can further define a set of services offered by a Helix Core server by using the `p4 server` and `p4 serverid` commands. Configuring the following servers require the use of a server spec:

- Commit server: central server in a distributed installation
- Edge server: node in a distributed installation
- Build server: replica that supports build farm integration
- Depot master: commit server with automated failover
- Depot standby: standby replica of the depot master
- Standby server: read-only replica that uses `p4 journalcopy`
- Forwarding standby: forwarding replica that uses `p4 journalcopy`

The `p4 serverid` command creates (or updates) a small text file named `server.id`. The `server.id` file always resides in a server's root directory.

The `p4 server` command can be used to maintain a list of all servers known to your installation. It can also be used to 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. The `p4 server` command can also be used to set a server's name (`P4NAME`).

Service users

There are three types of Perforce users: `standard` users, `operator` users, and `service` users. A `standard` user is a traditional user of Perforce, an `operator` user is intended for human or automated system administrators, and a `service` user is used for server-to-server authentication, as part of the replication process.

Service users are useful for remote depots in single-server environments, but are required for multi-server and distributed environments.

Create a `service` user for each master, replica, or proxy server that you control. Doing so greatly simplifies the task of interpreting your server logs. Service users can also help you improve security, by requiring that your edge servers and other replicas have valid login tickets before they can communicate with the master or commit server. Service users do not consume Perforce licenses.

A service user can run only the following commands:

- `p4 dbschema`
- `p4 export`
- `p4 login`
- `p4 logout`
- `p4 passwd`
- `p4 info`
- `p4 user`

To create a service user, run the command:

```
p4 user -f service1
```

The standard user form is displayed. Enter a new line to set the new user's `Type:` to be `service`; for example:

```
User:      service1
Email:     services@example.com
FullName:  Service User for Replica Server 1
Type:      service
```

By default, the output of `p4 users` omits service users. To include service users, run `p4 users -a`.

Tickets and timeouts for service users

A newly-created service user that is not a member of any groups is subject to the default ticket timeout of 12 hours. To avoid issues that arise when a service user's ticket ceases to be valid, create a group for your service users that features an extremely long timeout, or 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.

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 Perforce 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 a Helix Versioning Engine 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. The only valid startup command is `p4 pull`.

p4 pull vs. p4 replicate

Perforce 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.

For more information about `p4 replicate`, see "Perforce Metadata Replication" in the Perforce Knowledge Base:

http://answers.perforce.com/articles/KB_Article/Perforce-Metadata-Replication

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" above. The users' Perforce 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 Versioning Engine.

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.

Uses for replication

Here are some situations in which replica servers can be useful.

- For a failover or warm standby server, replicate both server metadata and versioned files by running two `p4 pull` commands in parallel. Each replica server requires one or more `p4 pull -u` instances to replicate versioned files, and a single `p4 pull` to replicate the metadata.

  If you are using `p4 pull` for both metadata and `p4 pull -u` for versioned files, start your replica server with `p4d -t protocol:host:port -Mreadonly -Dreadonly`. Commands that require read-only access to server metadata and to depot files will succeed. Commands that attempt to write to server metadata and/or depot files will fail gracefully.

  For a detailed example of this configuration, see "Configuring a read-only replica" on page 35.

- To configure an offline checkpointing or reporting server, only the master server's metadata needs to be replicated; versioned files do not need to be replicated.

  To use `p4 pull` for metadata-only replication, start the server with `p4d -t protocol:host:port -Mreadonly -Dnone`. You must specify a target. Do not configure the server to spawn any `p4 pull -u` commands that would replicate the depot files.

  In either scenario, commands that require read-only access to server metadata will succeed and commands that attempt to write to server metadata or attempt to access depot files will be blocked by the replica server.

Replication and protections

To apply the IP address of a replica user's workstation against the protections table, prepend the string `proxy-` to the workstation's IP address.

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

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

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

```
list    group    remotedev    192.168.10.0/24              -//...
list    group    remotedev    [2001:db8:16:81::]/48        -//...
write   group    remotedev    proxy-192.168.10.0/24         //...
write   group    remotedev    proxy-[2001:db8:16:81::]/48   //...
list    group    remotedev    proxy-10.0.0.0/8             -//...
list    group    remotedev    proxy-[2001:db8:1008::]/32   -//...
write   group    remotedev    10.0.0.0/8                    //...
write   group    remotedev    proxy-[2001:db8:1008::]/32    //...
```

The first line denies `list` access to all users in the `remotedev` group if they attempt to access Perforce 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 Core server directly from workstations on the central office's subnets. When visiting the local site, users from the `remotedev` group must access the Helix Core server directly.

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

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

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

How replica types handle requests

One way of explaining the differences between replica types is to describe how each type handles user requests; whether the server processes them locally, whether it forwards them, or whether it returns an error. The following table describes these differences.

- *Read only* commands include `p4 files`, `p4 filelog`, `p4 fstat`, `p4 user -o`
- *Work-in-progress* commands include `p4 sync`, `p4 edit`, `p4 add`, `p4 delete`, `p4 integrate`, `p4 resolve`, `p4 revert`, `p4 diff`, `p4 shelve`, `p4 unshelve`, `p4 submit`, `p4 reconcile`.
- *Global update* commands include `p4 user`, `p4 group`, `p4 branch`, `p4 label`, `p4 depot`, `p4 stream`, `p4 protect`, `p4 triggers`, `p4 typemap`, `p4 server`, `p4 configure`, `p4 counter`.

Replica type	Read-only commands	p4 sync, p4 client	Work-in-progress commands	Global update commands
Depot standby, standby, replica	Yes, local	Error	Error	Error
Forwarding standby, forwarding replica	Yes, local	Forward	Forward	Forward

Replica type	Read-only commands	p4 sync, p4 client	Work-in-progress commands	Global update commands
Build server	Yes, local	Yes, local	Error	Error
Edge server, workspace server	Yes, local	Yes, local	Yes, local	Forward
Standard server, depot master, commit server	Yes, local	Yes, local	Yes, local	Yes, local

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.

Note
Replication is asynchronous, and a replicated server is not recommended as the sole means of backup or disaster recovery. Maintaining a separate set of database checkpoints and depot backups (whether on tape, remote storage, or other means) is advised. Disaster recovery and failover strategies are complex and site-specific. Perforce Consultants are available to assist organizations in the planning and deployment of disaster recovery and failover strategies. For details, see:

http://www.perforce.com/services/consulting_overview

The following extended example configures a replica as a warm standby server for an existing Helix Versioning Engine 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, to avoid the risk of specifying an incorrect server root.

Master server setup

To define the behavior of the replica, you enter configuration information into the master server's `db.config` file using the `p4 configure set` command. Configure the master server first; its settings will be replicated to the replica later.

To configure the master, log in to Perforce as a superuser and perform the following steps:

1. To set the server named `Replica1` to use `master:11111` as the master server to pull metadata and versioned files, issue the command:

   ```
   $ p4 -p master:11111 configure set
   Replica1#P4TARGET=master:11111
   ```

   Perforce displays the following response:

   ```
   For server Replica1, configuration variable 'P4TARGET' set to
   'master:11111'
   ```

   > **Note**
   > To avoid confusion when working with multiple servers that appear identical in many ways, use the `-u` option to specify the superuser account and `-p` to explicitly specify the master Helix Core server's host and port.
   >
   > These options have been omitted from this example for simplicity. In a production environment, specify the host and port on the command line.

2. Set the `Replica1` server to save the replica server's log file using a specified file name. Keeping the log names unique prevents problems when collecting data for debugging or performance tracking purposes.

   ```
   $ p4 configure set Replica1#P4LOG=replica1Log.txt
   ```

3. Set the `Replica1` `server` configurable to `1`, which is equivalent to specifying the `-vserver=1` server startup option:

   ```
   $ p4 configure set Replica1#server=1
   ```

4. To enable process monitoring, set `Replica1`'s `monitor` configurable to `1`:

   ```
   $ p4 configure set Replica1#monitor=1
   ```

5. To handle the `Replica1` replication process, configure the following three `startup.n` commands. (When passing multiple items separated by spaces, you must wrap the entire set value in double quotes.)

   The first startup process sets `p4 pull` to poll once every second for journal data only:

   ```
   $ p4 configure set "Replica1#startup.1=pull -i 1"
   ```

   The next two settings configure the server to spawn two `p4 pull` threads at startup, each of which polls once per second for archive data transfers.

   ```
   $ p4 configure set "Replica1#startup.2=pull -u -i 1"
   $ p4 configure set "Replica1#startup.3=pull -u -i 1"
   ```

   Each `p4 pull -u` command creates a separate thread for replicating archive data. Heavily-loaded servers might require more threads, if archive data transfer begins to lag behind the replication of metadata. To determine if you need more `p4 pull -u` processes, read the contents of the `rdb.lbr` table, which records the archive data transferred from the master Helix Core server to the replica.

   To display the contents of this table when a replica is running, run:

   ```
   $ p4 -p replica:22222 pull -l
   ```

   Likewise, if you only need to know how many file transfers are active or pending, use `p4 -p replica:22222 pull -l -s`.

   If `p4 pull -l -s` indicates a large number of pending transfers, consider adding more `p4 pull -u startup.n` commands to address the problem.

   If a specific file transfer is failing repeatedly (perhaps due to unrecoverable errors on the master), you can cancel the pending transfer with `p4 pull -d -f file -r rev`, where file and rev refer to the file and revision number.

6. Set the `db.replication` (metadata access) and `lbr.replication` (depot file access) configurables to readonly:

   ```
   $ p4 configure set Replica1#db.replication=readonly
   $ p4 configure set Replica1#lbr.replication=readonly
   ```

   Because this replica server is intended as a warm standby (failover) server, both the master server's metadata and its library of versioned depot files are being replicated. When the replica is running, users of the replica will be able to run commands that access both metadata and the server's library of depot files.

7. Create the service user:

   ```
   $ p4 user -f service
   ```

   The user specification for the `service` user opens in your default editor. Add the following line to the user specification:

   ```
   Type: service
   ```

   Save the user specification and exit your default editor.

   By default, the service user is granted the same 12-hour login timeout as standard users. To prevent the service user's ticket from timing out, create a group with a long timeout on the master server. In this example, the `Timeout:` field is set to two billion seconds, approximately 63 years:

   ```
   $ p4 group service_group
   ```

   ```
   Users: service
   Timeout: 2000000000
   ```

   For more details, see "Tickets and timeouts for service users" on page 30.

8. Set the service user protections to `super` in your protections table. (See "Permissions for service users" on page 30.) It is good practice to set the security level of all your Helix Core Servers to at least 1 (preferably to 3, so as to require a strong password for the service user, and ideally to 4, to ensure that *only* authenticated service users may attempt to perform replica or remote depot transactions.)

   ```
   $ p4 configure set security=4
   $ p4 passwd
   ```

9. Set the `Replica1` configurable for the `serviceUser` to `service`.

   ```
   $ p4 configure set Replica1#serviceUser=service
   ```

   This step configures the replica server to authenticate itself to the master server as the `service` user; this is equivalent to starting `p4d` with the `-u service` option.

10. If the user running the replica server does not have a home directory, or if the directory where the default `.p4tickets` file is typically stored is not writable by the replica's Helix Core server process, set the replica `P4TICKETS` value to point to a writable ticket file in the replica's Helix Core server root directory:

    ```
    $ p4 configure set
    "Replica1#P4TICKETS=/p4/replica/.p4tickets"
    ```

Creating the replica

To configure and start a replica server, perform the following steps:

1. Boot-strap the replica server by checkpointing the master server, and restoring that checkpoint to the replica:

   ```
   $ p4 admin checkpoint
   ```

   (For a new setup, we can assume the checkpoint file is named `checkpoint.1`)

2. Move the checkpoint to the replica server's `P4ROOT` directory and replay the checkpoint:

   ```
   $ p4d -r /p4/replica -jr $P4ROOT/checkpoint.1
   ```

3. Copy the versioned files from the master server to the replica.

   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 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. To create a valid ticket file, use `p4 login` to connect to the master server and obtain a ticket on behalf of the replica server's service user. On the machine that will host the replica server, run:

   ```
   $ p4 -u service -p master:11111 login
   ```

   Then move the ticket to the location that holds the `P4TICKETS` file for the replica server's service user.

At this point, your replica server is configured to contact the master server and start replication. Specifically:

- A service user (`service`) in a group (`service_group`) with a long ticket timeout
- A valid ticket for the replica server's service user (from `p4 login`)
- A replicated copy of the master server's `db.config`, holding the following preconfigured settings applicable to any server with a `P4NAME` of `Replica1`, specifically:
  - A specified service user (named `service`), which is equivalent to specifying `-u service` on the command line
  - A target server of `master:11111`, which is equivalent to specifying `-t master:11111` on the command line
  - Both `db.replication` and `lbr.replication` set to `readonly`, which is equivalent to specifying `-M readonly -D readonly` on the command line
  - A series of `p4 pull` commands configured to run when the master server starts

Starting the replica

To name your server `Replica1`, set `P4NAME` or specify the `-In` option and start the replica as follows:

```
$ p4d -r /p4/replica -In Replica1 -p replica:22222 -d
```

When the replica starts, all of the master server's configuration information is read from the replica's copy of the `db.config` table (which you copied earlier). The replica then spawns three `p4 pull` threads: one to poll the master server for metadata, and two to poll the master server for versioned files.

> **Note**
> The `p4 info` command displays information about replicas and service fields for untagged output as well as tagged output.

Testing the replica

Testing p4 pull

To confirm that the `p4 pull` commands (specified in `Replica1`'s `startup.`*n* configurations) are running, issue the following command:

```
$ p4 -u super -p replica:22222 monitor show -a
18835 R service00:04:46 pull -i 1
18836 R service00:04:46 pull -u -i 1
18837 R service00:04:46 pull -u -i 1
18926 R super 00:00:00 monitor show -a
```

If you need to stop replication for any reason, use the `p4 monitor terminate` command:

```
$ p4 -u super -p replica:22222 monitor terminate 18837 process
'18837' marked for termination
```

To restart replication, either restart the Helix Core server process, or manually restart the replication command:

```
$ p4 -u super -p replica:22222 pull -u -i 1
```

If the `p4 pull` and/or `p4 pull -u` processes are terminated, read-only commands will continue to work for replica users as long as the replica server's `p4d` is running.

Testing file replication

Create a new file under your workspace view:

```
$ echo "hello world" > myfile
```

Mark the file for add:

```
$ p4 -p master:11111 add myfile
```

And submit the file:

```
$ p4 -p master:11111 submit -d "testing replication"
```

Wait a few seconds for the pull commands on the replica to run, then check the replica for the replicated file:

```
$ p4 -p replica:22222 print //depot/myfile
//depot/myfile#1 - add change 1 (text)
hello world
```

If a file transfer is interrupted for any reason, and a versioned file is not present when requested by a user, the replica server silently retrieves the file from the master.

> **Note**
> Replica servers in `-M readonly -D readonly` mode will retrieve versioned files from master servers even if started without a `p4 pull -u` command to replicate versioned files to the replica. Such servers act as "on-demand" replicas, as do servers running in `-M readonly -D ondemand` mode or with their `lbr.replication` configurable set to `ondemand`.
>
> *Administrators*: be aware that creating an on-demand replica of this sort can still affect server performance or resource consumption, for example, if a user enters a command such as `p4 print //...`, which reads every file in the depot.

Verifying the replica

When you copied the versioned files from the master server to the replica server, you relied on the operating system to transfer the files. To determine whether data was corrupted in the process, run `p4 verify` on the replica server:

```
$ p4 verify //...
```

Any errors that are present on the replica but not on the master indicate corruption of the data in transit or while being written to disk during the original copy operation. (Run `p4 verify` on a regular basis, because a failover server's storage is just as vulnerable to corruption as a production server.)

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 Versioning Engine 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 Perforce 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 Versioning Engine is appropriate to the task at hand.

For an example of such a configuration, see "Using P4Broker With Replica Servers" in the Perforce Knowledge Base:

http://answers.perforce.com/articles/KB_Article/Using-P4Broker-With-Replica-Servers

For more information about the Helix Broker, see "The Helix Broker" on page 77.

Upgrading replica servers

It is best practice to upgrade any server instance replicating from a master server first. 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.

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), it is recommended to wait for all archive transfers to end before shutting down the replica and commencing the upgrade. You must manually delete the `rdb.lbr` file in the replica server's root before restarting the replica.

For more information, see "Upgrading Replica Servers" in the Perforce Knowledge Base:

http://answers.perforce.com/articles/KB_Article/Upgrading-Replica-Servers/

Configuring a forwarding replica

A forwarding replica offers a blend of the functionality of the Helix Proxy with the improved performance of a replica. The following considerations are relevant:

The Helix Proxy is easier to configure and maintain, but caches only file content; it holds no metadata. A forwarding replica caches both file content and metadata, and can therefore process many commands without requesting additional data from the master server. This behavior enables a forwarding replica to offload more tasks from the master server and provides improved performance. The trade-off is that a forwarding replica requires a higher level of machine provisioning and administrative considerations compared to a proxy.

A read-only replica rejects commands that update metadata; a forwarding replica does not reject such commands, but forwards them to the master server for processing, and then waits for the metadata update to be processed by the master server and returned to the forwarding replica. Although users connected to the forwarding replica cannot write to the replica's metadata, they nevertheless receive a consistent view of the database.

If you are auditing server activity, each of your forwarding replica servers must have its own `P4AUDIT` log configured.

Configuring the master server

The following example assumes an environment with a regular server named `master`, and a forwarding replica server named `fwd-replica` on a host named `forward`.

1. Start by configuring a read-only replica for warm standby; see "Configuring a read-only replica" on page 35 for details. (Instead of `Replica1`, use the name `fwd-replica`.)

2. On the master server, configure the forwarding replica as follows:

```
$ p4 server fwd-1667
```

The following form is displayed:

```
ServerID:       fwd-1667
Name:           fwd-replica
Type:           server
Services:       forwarding-replica
Address:        tcp:forward:1667
Description:
        Forwarding replica pointing to master:1666
```

Configuring the forwarding replica

1. On the replica machine, assign the replica server a serverID:

```
$ p4 serverid fwd-1667
```

When the replica server with the `serverID:` of `fwd-1667` (which was previously assigned the `Name:` of `fwd-replica`) pulls its configuration from the master server, it will behave as a forwarding replica.

2. On the replica machine, restart the replica server:

```
$ p4 admin restart
```

Configuring a build farm server

Continuous integration and other similar development processes can impose a significant workload on your Perforce infrastructure. Automated build processes frequently access the Perforce 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 build farm server, you can offload the workload of the automated build processes to a separate machine, and ensure that your main Helix Core server's resources are available to your users for their normal day-to-day tasks.

> **Note**
> Build farm servers were implemented in Helix Core 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 farm server. As discussed in "Commit-edge Architecture" on page 56, edge servers offer all the functionality of build farm servers and yet offload more work from the main server and improve performance, with the additional flexibility of being able to run write commands as part of the build process.

A Helix Versioning Engine intended for use as a build farm must, by definition:

- Permit the creation and configuration of client workspaces
- Permit those workspaces to be synced

One issue with implementing a build farm rather than a read-only replica is that under Helix Core, both of those operations involve writes to metadata: in order to use a client workspace in a build environment, the workspace must contain some information (even if nothing more than the client workspace root) specific to the build environment, and in order for a build tool to efficiently sync a client workspace, a build server must be able to keep some record of which files have already been synced.

To address these issues, build farm replicas host their own local copies of certain metadata: in addition to the Helix Core commands supported in a read-only replica environment, build farm replicas 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 farm replica servers must have its own `P4AUDIT` log configured.

Configuring the master server

The following example assumes an environment with a regular server named `master`, and a build farm replica server named `buildfarm1` on a host named `builder`.

1. Start by configuring a read-only replica for warm standby; see "Configuring a read-only replica" on page 35 for details. (That is, create a read-only replica named `buildfarm1`.)
2. On the master server, configure the master server as follows:

   ```
   $ p4 server master-1666
   ```

   The following form is displayed:

```
# A Perforce Server Specification.
#
#  ServerID:    The server identifier.
#  Type:        The server type: server/broker/proxy.
#  Name:        The P4NAME used by this server (optional).
#  Address:     The P4PORT used by this server (optional).
#  Description: A short description of the server (optional).
#  Services:    Services provided by this server, one of:
#          standard: standard Perforce server
#          replica: read-only replica server
#          broker: p4broker process
#          proxy: p4p caching proxy
#          commit-server: central server in a distributed installation
#          edge-server: node in a distributed installation
```

```
#          forwarding-replica: replica which forwards update commands
#          build-server: replica which supports build automation
#          P4AUTH: server which provides central authentication
#          P4CHANGE: server which provides central change numbers
#
# Use 'p4 help server' to see more about server ids and services.

ServerID:       master-1666
Name:           master-1666
Type:           server
Services:       standard
Address:        tcp:master:1666
Description:
        Master server - regular development work
```

1. Create the master server's `server.id` file. On the master server, run the following command:

   ```
   $ p4 -p master:1666 serverid master-1666
   ```

2. Restart the master server.

   On startup, the master server reads its server ID of `master-1666` from its `server.id` file. It takes on the `P4NAME` of `master` and uses the configurables that apply to a `P4NAME` setting of `master`.

Configuring the build farm replica

1. On the master server, configure the build farm replica server as follows:

   ```
   $ p4 server builder-1667
   ```

   The following form is displayed:

   ```
   ServerID:       builder-1667
   Name:           builder-1667
   Type:           server
   Services:       build-server
   Address:        tcp:builder:1667
   Description:
           Build farm - bind workspaces to builder-1667
           and use a port of tcp:builder:1667
   ```

2. Create the build farm replica server's `server.id` file. On the *replica* server (not the master server), run the following command

```
$ p4 -p builder:1667 serverid builder-1667
```

3. Restart the replica server.

   On startup, the replica build farm server reads its server ID of `builder-1667` from its `server.id` file.

   Because the server registry is automatically replicated from the master server to all replica servers, the restarted build farm server takes on the `P4NAME` of `buildfarm1` and uses the configurables that apply to a `P4NAME` setting of `buildfarm1`.

   In this example, the build farm server also acknowledges the `build-server` setting in the `Services:` field of its `p4 server` form.

Binding workspaces to the build farm replica

At this point, there should be two servers in operation: a master server named `master`, with a server ID of `master-1666`, and a `build-server` replica named `buildfarm1`, with a server ID of `builder-1667`.

1. Bind client workspaces to the build farm 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 replica server, create a client workspace with `p4 client`:

```
$ p4 -c build0001 -p builder:1667 client build0001
```

   When creating a new workspace on the build farm replica, you must ensure that your current client workspace has a ServerID that matches that required by `builder: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. For more information, see:

   http://answers.perforce.com/articles/KB_Article/Build-Farm-Client-Management

   When the `p4 client` form appears, set the `ServerID:` field to `builder-1667`.

2. Sync the bound workspace

   Because the client workspace `build0001` is bound to `builder-1667`, users on the master server are unaffected, but users on the build farm server are not only able to edit its specification, they are able to sync it:

   ```
   $ export P4PORT=builder:1667
   $ export P4CLIENT=build0001
   $ p4 sync
   ```

   The replica's have list is updated, and does not propagate back to the master. Users of the master server are unaffected.

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 farm 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.

In a real-world scenario, performance on the master would likely improve for all users, as the number of read and write operations on the master server's database would be substantially reduced.

If there are database tables that you know your build farm replica does not require, consider using the `-F` and `-T` filter options to `p4 pull`. Also consider specifying the `ArchiveDataFilter:`, `RevisionDataFilter:` and `ClientDataFilter:` fields of the replica's `p4 server` form.

If your automation load should exceed the capacity of a single machine, you can configure additional build farm servers. There is no limit to the number of build farm servers that you may operate in your installation.

Configuring a replica with shared archives

Normally, a Perforce replica service retrieves its metadata and file archives on the user-defined pull interval, for example `p4 pull -i 1`. When the `lbr.replication` configurable is set to `ondemand` or `shared` (the two are synonymous), metadata is retrieved on the pull interval and archive files are retrieved only when requested by a client; new files are not automatically transferred, nor are purged files removed.

When a replica server is configured to directly share the same physical archive files as the master server, whether the replica and master are running on the same machine or via network shared storage, the replica simply accesses the archives directly without requiring the master to send the archives files to the replica. This can form part of a High Availability configuration.

> **Warning**
> When archive files are directly shared between a replica and master server, the replica *must* have `lbr.replication` set to `ondemand` or `shared`, or archive corruption may occur.

To configure a replica to share archive files with a master, perform the following steps:

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 for details.

2. If you have not already done so, configure the replica server as a forwarding replica.

   See "Configuring a read-only replica" on page 35.

3. Set `lbr.replication`.

   For example: `p4 configure set REP13-1#lbr.replication=ondemand`

4. Restart the replica, specifying the share archive location for the replica's root.

Once these steps have been completed, the following conditions are in effect:

- archive file content is only retrieved when requested, and those requests are made against the shared archives.
- no entries are written to the `rdb.lbr` librarian file during replication.
- commands that would schedule the transfer of file content, such as `p4 pull -u` and `p4 verify -t` are rejected:

```
$ p4 pull -u
This command is not used with an ondemand replica server.
$ p4 verify -t //depot/...
This command is not used with an ondemand replica server.
```

- if startup configurables, such as `startup.N=pull -u`, are defined, the replica server attempts to run such commands. Since the attempt to retrieve archive content is rejected, the replica's server log will contain the corresponding error:

```
Perforce server error:
        2014/01/23 13:02:31 pid 6952 service-od@21131 background
'pull -u -i 10'
        This command is not used with an ondemand replica server.
```

Filtering metadata during replication

As part of an HA/DR solution, one typically wants to ensure that all the metadata and all the versioned files are replicated. In most other use cases, particularly build farms 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 in (or out) 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 offices where other projects are being developed, and build farms don't require access to the endless stream of changes to office documents and spreadsheets associated with a typical large enterprise.

The simplest way to filter metadata is by using the `-T tableexcludelist` option with `p4 pull` command. If you know, for example, that a build farm 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 Perforce command operations, and furthermore, offers no means of control over which versioned files are replicated.

You can gain much more fine-grained control over what data is replicated by using the `ClientDataFilter:`, `RevisionDataFilter:`, and `ArchiveDataFilter:` fields of the `p4 server` form. These options enable you to replicate (or exclude from replication) those portions of your server's metadata and versioned files that are of interest at the replica site.

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 must specify the ServerID associated with the server spec that holds the filters:

```
$ p4 configure set "site1-1668#startup.1=pull -i 30 -P site1-1668"
```

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 farm replica scenario. The ongoing work of the organization (whether it be code, business documents, or the latest video commercial) 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 at its immediate disposal:

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 must specify the ServerID associated with the server spec that holds the filters:

```
$ p4 configure set "builder-1669#startup.1=pull -i 30 -P
builder-1669"
```

The `p4 pull` thread that pulls metadata for replication filters out 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 ones of interest to a build farm. The only metadata that is available is that which concerns released code. All released code is automatically transferred to the build farm before any requests are made, so that when the build farm performs a `p4 sync`, the sync is performed locally.

Verifying replica integrity

Tools to ensure data integrity in 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, then proceed to 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, newly defined triggers allow you to take some custom action when journals are rotated. For more information, see the section "Triggering on journal rotation" in *Helix Versioning Engine 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 `rpl.checksum.auto`, `rpl.checksum.change`, and `rpl.checksum.table` configurables to the desired levels of integrity checking. 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.group`, `db.groupx`, `db.integed`, `db.integtx`, `db.ldap`, `db.protect`, `db.rev`, `db.revcx`, `db.revdx`, `db.revhx`, `db.revtx`, `db.stream`, `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.ixtext`, `db.ixtexthx`, `db.job`, `db.logger`, `db.message`, `db.nameval`, `db.property`, `db.remote`, `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.
`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. The examples in the section "Configuring a build farm server" on page 44 illustrate the use of manually-assigned names that are easy to remember, but in very large environments, there may be more servers in a build farm 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. Perforce 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 Architecture

Commit-edge architecture is a specific replication configuration. It is a good solution for geographically distributed work groups, and it offers significant performance advantages. At a minimum it is made up of the following kinds of servers:

- A *commit* server that stores the canonical archives and permanent metadata. In working terms, it is similar to a Perforce master server, but might not contain all workspace information.
- An *edge* server that contains a replicated copy of the commit server data and a unique, local copy of some workspace and work-in-progress information. It can process read-only operations and operations like `p4 edit` that only write to the local data. In working terms, it is similar to a forwarding replica, but contains local workspace data and can handle more operations with no reliance on the commit server. You can connect multiple edge servers to a commit server.

Since an edge server can handle most routine operations locally, the edge-commit architecture offloads a significant amount of processing work from the commit server and reduces data transmission between commit and edge servers. This greatly improves performance.

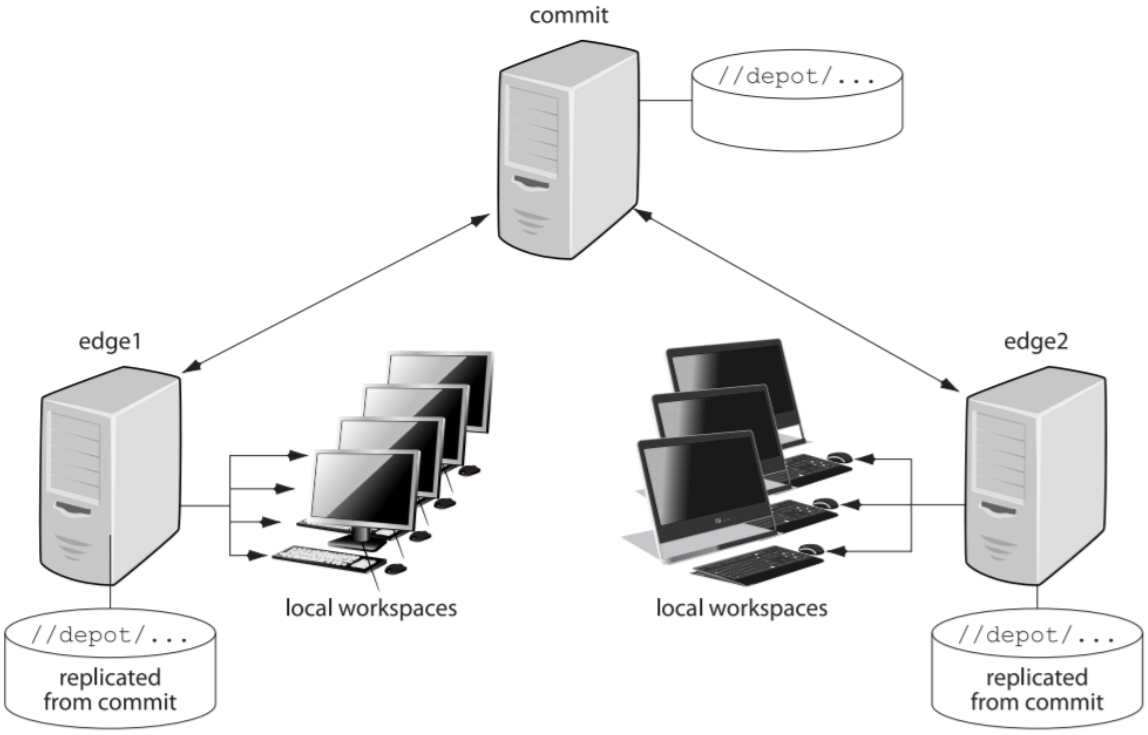
From a user's perspective, most typical operations until the point of submit are handled by an edge server. As with a forwarding replica, read operations, such as obtaining a list of files or viewing file history, are local. In addition, with an edge server, syncing, checking out, merging, resolving, and reverting files are also local operations.

> **Note**
> You may not issue the `p4 unsubmit` and `p4 resubmit` commands to an edge server. You may only issue these commands to a commit server.

Commit-edge architecture builds upon Perforce replication technology. You should read "Perforce Replication" on page 21 before attempting to deploy a commit-edge configuration.

An edge server can be used instead of a build farm server, and this usage is referred to as a *build edge server*. If the only users of an edge server are build processes, then your backup (disaster recovery) strategy may be simplified as you do not need to backup the local edge server-specific workspace and related information. See "Migrating from existing installations" on page 64.

The next figure illustrates one possible commit-edge configuration: a commit server, communicating with two edge-servers, each of which handles multiple workspaces.



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 Core 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, which is described in detail in the following sections includes the following major steps:

1. On the commit server: Create a service user account for each edge server you plan to create.
2. On the commit server: Create commit and edge server configurations.
3. Create and start the edge server.

You must have `super` privileges to perform these steps.

Create a service user account for the edge server

To support secure communication between the commit server and the edge server, a user account of type service must be created. In the example below, we use a unique service user name for the tokyo edge server, but one could also use a generic service user name and use it for multiple edge servers.

1. Create the service user account.

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

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

2. Add the service user to a group with an unlimited timeout. This prevents the service user login from the edge server from timing out.

   ```
   $ p4 group no_timeout
   ```

   In the `group` spec, set the `Users:` field to `svc_tokyo_edge` and the `Timeout:` field to `unlimited`.

3. Assign a password to the service user by providing a value at the prompt.

   ```
   $ p4 passwd svc_tokyo_edge
   ```

4. Assign the `svc_tokyo_edge` service user `super` protections in the protect spec.

   ```
   $ p4 protect
   super user svc_tokyo_edge * //...
   ```

Create commit and edge server configurations

The following steps are needed to configure the commit and edge servers.

> **Note**
> It is best to set the **P4NAME** and ServerID to the same value: this makes it easy to isolate configuration variables on each server in a distributed environment.

1. Create the commit server specification:

   ```
   $ p4 server chicago_commit
   ```

   In the server spec, set the `Services:` field to `commit-server` and the `Name:` field to `chicago_commit`.

2. Create the edge server specification:

   ```
   $ p4 server tokyo_edge
   ```

   In the server spec, set the `Services:` field to `edge-server` and the `Name:` field to `tokyo_edge`.

3. Set the server ID of the commit server:

```
$ p4 serverid chicago_commit
```

4. This step, which sets the `journalPrefix` value on the commit and edge server to control the name and location of server checkpoints and rotated journals, is not required, but it is a best practice. During the replication process, the edge server might need to locate the rotated journal on the commit server; having `journalPrefix` defined on the commit server allows the edge server to easily identify the name and location of rotated journals:

```
$ p4 configure set chicago_
commit#journalPrefix=/chicago/backup/p4d_backup
$ p4 configure set tokyo_
edge#journalPrefix=/tokyo/backup/p4d_backup
```

5. Set `P4TARGET` for the edge server to identify the commit server:

```
$ p4 configure set tokyo_
edge#P4TARGET=chicago.perforce.com:1666
```

6. Set the service user in the edge server configuration:

```
$ p4 configure set tokyo_edge#serviceUser=svc_tokyo_edge
```

7. Set the location for the edge server's log files:

```
$ p4 configure set tokyo_edge#P4LOG=/tokyo/logs/tokyo_
edge.log
```

8. Set `P4TICKETS` location for the service user in the edge and commit server configuration:

```
$ p4 configure set chicago_
commit#P4TICKETS=/chicago/p4root/.p4tickets
$ p4 configure set tokyo_
edge#P4TICKETS=/tokyo/p4root/.p4tickets
```

9. Configure the edge server database and archive nodes:

```
$ p4 configure set tokyo_edge#db.replication=readonly
$ p4 configure set tokyo_edge#lbr.replication=readonly
```

10. Define startup commands for the edge server to periodically pull metadata and archive data.

```
$ p4 configure set tokyo_edge#startup.1="pull -i 1"
$ p4 configure set tokyo_edge#startup.2="pull -u -i 1"
$ p4 configure set tokyo_edge#startup.3="pull -u -i 1"
```

These commands configure three *pull threads*, each attempting to fetch metadata once per second. The latter two pull threads use the `-u` option to transfer archive files instead of journal records. Typically, there is more file data to transfer than metadata, so one pull thread fetches the journal data, and two fetch the file data.

Create and start the edge server

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, but 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.resolve,db.locks,
        db.revsh,db.workingx,db.resolvex" -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. Create the service user login ticket in the location specified in the edge configuration:

```
$ p4 -E P4TICKETS=/chicago/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 additional information on copying files, see:

   - http://answers.perforce.com/articles/KB/2558
   - "Creating the replica" on page 38

6. Start the edge server using syntax appropriate for your platform.

   For example:

   ```
   $ p4d -r /tokyo/p4root -d
   ```

   Consult the following sources for detailed instructions for UNIX and Windows, which appear in the "Installing and Upgrading the Server" chapter of the *Helix Versioning Engine Administrator Guide: Fundamentals*.

7. Check the status of replication by running the following command against the edge server.

   ```
   $ p4 pull -lj
   ```

8. Create the service user login ticket from the commit to the edge server. On the commit server:

   ```
   $ p4 -E P4TICKETS=/chicago/p4root/.p4tickets -u svc_tokyo_
   edge
       -p tokyo.perforce.com:1666 login
   ```

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 commands you want executed 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 -l 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
```

Setting global client views

The `server.global.client.views` configurable determines whether the view maps of a non-stream client on an edge server or workspace 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 or workspace 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" below 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 66 provides instructions for different migration scenarios.

Replacing existing proxies and replicas

If you currently use Perforce proxies, evaluate whether these should be replaced with edge servers. 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 68 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 in this section, we recommend you contact Perforce support for assistance 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
        ```

        See the "Helix Versioning Engine Reference" appendix in the *Helix Versioning Engine Administrator Guide: Fundamentals*, for options that can be used to produce a filtered journal dump file, specifically 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 is pretty simple:

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!

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. This section describes these issues.

- 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 74 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 network 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
  ```

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" below.

You can view a shelf's promotion status using the `-ztag` output of the `p4 describe`, `p4 changes`, or `p4 change -o` commands.

See "Working with promoted shelves" on the next page for more information on the limitations of working on promoted shelves.

Automatically promoting shelves

When the edge server and commit server are configured to access the same archive contents, shelf promotion occurs automatically, and promoting shelved 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: `dm.shelve.promote=1`.

  This makes edge servers always 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. (Generally, it is a bad idea to enable automatic promotion because it causes a lot of unnecessary file transfers for shelved files that are not meant to be shared.)
- 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`, the process works as follows:

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 new `-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

The following limitations apply when working with promoted shelves:

- Once a shelf is promoted, it stays promoted.

  There is no mechanism to *unpromote* a shelved changelist; instead, delete the shelved files from the changelist.
- You may unshelve a promoted shelf into open files and branches on a server from where the shelf did not originate.
- You cannot unshelve a remote promoted shelf into already-open local files.
- You cannot unload an edge server workspace if you have promoted shelves.
- You can run `p4 submit -e` on a promoted shelf only on the server that owns the change.
- You can 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 *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 Versioning Engine 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.

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 *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`. They are described in the following table.

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. As `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 his 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. Therefore, an edge server may 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. Specifically, you need to do the following:

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 may need some time 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.

- **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 Perforce, 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 Perforce functionality while isolating build workspace metadata. Using an edge server in this way is similar to using a build farm replica, 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 may 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. Since the commit server does not verify that the unload depot is empty on every edge server, you must specify `p4 depot -d -f` in order to delete the unload depot from the commit server.

- **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 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.

The Helix Broker

The Helix Broker (P4Broker) enables you to implement local policies in your Perforce environment by allowing you to restrict the commands that can be executed, or redirect specific commands to alternate (replica or edge) Helix Core servers.

The Helix Broker is a server process that mediates between Perforce client applications and Helix Core servers, including proxy servers. For example, Perforce client applications can connect to a proxy server that connects to the broker, which then connects to a Helix Core server. Or, Perforce client applications can connect to a broker configured to redirect reporting-related commands to a read-only replica server, while passing other commands through to a master server. You can use a broker to solve load-balancing, security, or other issues that can be resolved by sorting requests directed to one or more Helix Core servers.

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 Core 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 backup 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 Versioning Engine.

System requirements

To use the Helix Broker, you must have:

- A Helix Core server at release 2007.2 or higher (2012.1 or higher to use SSL).
- Perforce 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 Versioning Engine (`p4d`), preferably on the same machine.

Installing the broker

To install P4Broker, do the following:

1. Download the `p4broker` executable from the Perforce website,
2. Copy it to a suitable directory on the host (such as `/usr/local/bin`), and ensure that the binary is executable:

```
$ chmod +x p4broker
```

Running the broker

After you have created your configuration file (see "Configuring the broker" on page 82), 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 "Installing P4Broker on Windows and Unix systems" in the Perforce Knowledge Base:

http://answers.perforce.com/articles/KB_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 Versioning Engine. See "Enabling SSL support" on page 32. The users' Perforce applications must be configured to trust the fingerprint of the broker.

To encrypt the connection between a Helix Broker and a Helix Versioning Engine, your broker must be configured so as to trust the fingerprint of the Helix Versioning Engine. 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 Versioning Engine, and must set `P4TRUST`, specifying the path to that file (`P4TRUST` cannot be specified in the broker configuration file).

For complete information about enabling SSL for the broker, see:
http://answers.perforce.com/articles/KB/2596

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 Perforce 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 Perforce server directly from workstations on the central office's subnets. When visiting the local site, users from the `remotedev` group must access the Perforce server directly.

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

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

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

P4Broker options

Option	Meaning
`-c` *`file`*	Specify a configuration file. Overrides `P4BROKEROPTIONS` setting.
`-C`	Output a sample configuration file, and then exit.
`-d`	Run as a daemon (in the background).
`-f`	Run as a single-threaded (non-forking) process.
`-h`	Print help message, and then exit.
`-q`	Run quietly (no startup messages).
`-V`	Print broker version, and then exit.

Option	Meaning
`-v subsystem=level`	Set server trace options. Overrides the value of the `P4DEBUG` setting, but does *not* override the `debug-level` setting in the `p4broker.conf` file. Default is null. The server command trace options and their meanings are as follows. ▪ `server=0` Disable broker command logging. ▪ `server=1` Logs broker commands to the server log file. ▪ `server=2` In addition to data logged at level `1`, logs broker command completion and basic information on CPU time used. Time elapsed is reported in seconds. On UNIX, CPU usage (system and user time) is reported in milliseconds, as per `getrusage()`. ▪ `server=3` In addition to data logged at level 2, adds usage information for compute phases of `p4 sync` and `p4 flush` (`p4 sync -k`) commands. For command tracing, output appears in the specified log file, showing the date, time, username, IP address, and command for each request processed by the server.
`-Gc`	Generate SSL credentials files for the broker: create a private key (`privatekey.txt`) and certificate file (`certificate.txt`) in `P4SSLDIR`, and then exit. Requires that `P4SSLDIR` be set to a directory that is owned by the user invoking the command, and that is readable only by that user. If `config.txt` is present in `P4SSLDIR`, generate a self-signed certificate with specified characteristics.
`-Gf`	Display the fingerprint of the broker's public key, and exit. Administrators can communicate this fingerprint to end users, who can then use the `p4 trust` command to determine whether or not the fingerprint (of the server to which they happen to be connecting) is accurate.

Configuring the broker

P4Broker is controlled by a broker configuration file. The broker configuration file is a text file that contains rules for:

- Specifying which commands that individual users can use.
- Defining commands that are to be redirected to 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 three 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 the command

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 Perforce 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.

Protocol	Behavior
`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.

Global settings

The following settings apply to all operations you specify for the broker.

Setting	Meaning	Example
`target`	The default Helix Versioning Engine (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 Perforce 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`*`;`

Setting	Meaning	Example
`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 Perforce Administrator. This is displayed in certain error messages.	`admin-name = "P4 Admin";`
`admin-email`	An email address where users can contact their Perforce Administrator. This address is displayed to users when broker configuration problems occur.	`admin-email = admin@example.com;`
`admin-phone`	The telephone number of the Perforce 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 92. Multiple `altserver` settings may appear in the broker configuration file, one for each alternate server.	`altserver` *name* `{ target=` [*protocol*`:]`*host*`:`*port* `};` Each `altserver` setting must appear on one line.

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 Versioning Engine 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 Perforce client workspace setting in effect when the command was issued.
`prog`	The Perforce 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 Perforce 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`.
`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 page 90.
`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.

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, etc.

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
`clientProg:`	client application program
`clientVersion:`	version of client application program
`clientProtocol:`	level of client protocol
`apiProtocol:`	level of api protocol
`maxLockTime:`	maximum lock time (in ms) to lock tables before aborting
`maxScanRows:`	maximum number of rows of data scanned by a command
`maxResults:`	maximum number of rows of result data to be returned
`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 next 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.

Action	Definition
`CONTINUE`	Defer to the next command handler matching a given command. For additional information on using multiple handlers, see: http://answers.perforce.com/articles/KB/11309

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.

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 "Perforce Replication" on page 21. 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 as follows:

```
altserver name { target=[protocol:]host:port }
```

The name assigned to the alternate server is used in command handler specifications. See "Command handler specifications" on page 87.

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 Perforce 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 17.

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).

Using the broker as a load-balancing router

Previous sections described how you can use the broker to direct specific commands to specific servers. You can also configure the broker to act as a load-balancing router. When you configure a broker to act as a router, Perforce builds a `db.routing` table that is processed by the router to determine which server an incoming command should be routed to. (The `db.routing` table provides a mapping of clients and users to their respective servers. To reset the `db.routing` table, remove the `db.routing` file.)

This section explains how you configure the broker to act as a router, and it describes routing policy and behavior.

Configuring the broker as a router

To configure the broker as a router, add the `router` statement to the top level of the broker configuration. The target server of the broker-router should be a commit or master server.

```
target      = commitserv.example.com:1666;
listen      = 1667;
directory   = /p4/broker;
logfile     = broker.log;
debug-level = server=1;
admin-name  = "Perforce Admins";
admin-phone = 999/911;
admin-email = perforce-admins@example.com;
router;
```

You must then include `altserver` statements to specify the edge servers of the commit server (or the workspace servers of the depot master) that is the target server of the broker-router.

```
altserver: edgeserv1
{
    target = edgeserve1.example.com:1669;
}
```

If you are using the broker to route messages for a commit-edge architecture, you must list all existing edge serves as altservers.

Routing policy and behavior

When a command arrives at the broker, the broker looks in its `db.routing` table to determine where the command should be routed. The routing logic attempts to bind a user to a server where they already have clients. You can modify the routing choice on the `p4` command line using the following argument to override routing for that command.

```
-Zroute=serverID
```

Routing logic uses a default destination server, chosen at random, if a client and user have no binding to an existing edge server. That is, requests are routed to an existing altserver that is randomly chosen unless a specific destination is given.

- To route requests to the commit server, use the destination form as follows:

```
target      = commitserv.example.com:1666;
listen      = 1667;
directory   = /p4/broker;
logfile     = broker.log;
debug-level = server=1;
admin-name  = "Perforce Admins";
admin-phone = 999/911;
admin-email = perforce-admins@example.com;

router;
destination target;
```

- To cause new users to be bound to a new edge server rather than being assigned to existing edge servers, use a destination form like the following:

```
target      = commitserv.example.com:1666;
listen      = 1667;
directory   = /p4/broker;
logfile     = broker.log;
debug-level = server=1;
admin-name  = "Perforce Admins";
admin-phone = 999/911;
admin-email = perforce-admins@example.com;

router;
destination = "myNewEdge";
```

- To force a command to be routed to the commit server, use an `action = redirect` rule with a `destination target` statement; for example:

```
command: regex pattern
{
    action=redirect;
    destination target;
}
```

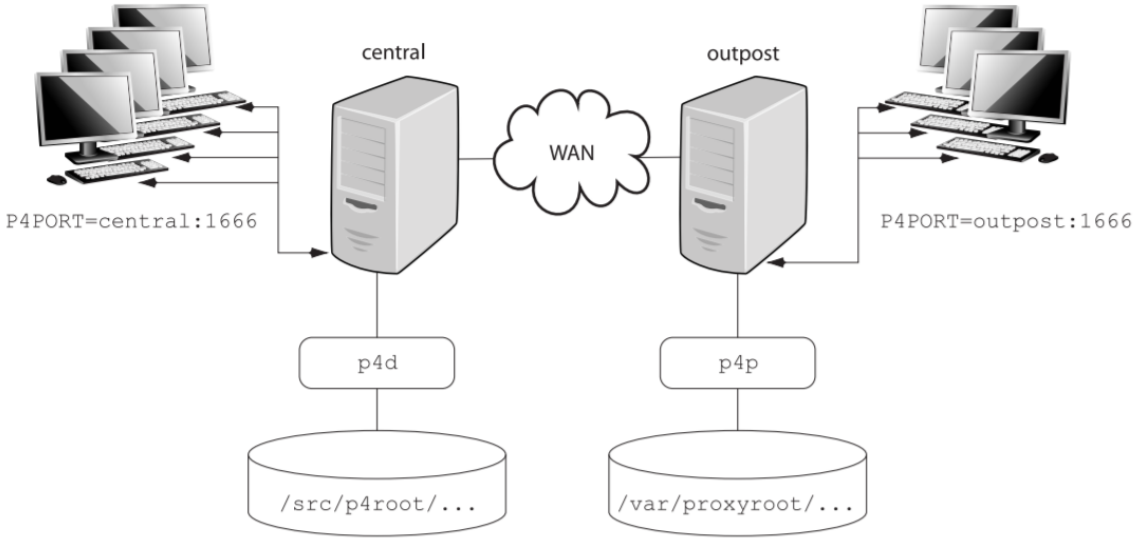
> **Note**
> You need to remove the `db.routing` file when you move clients to a different workspace or edge server.

Helix Proxy

Perforce is built to handle distributed development in a wide range of network topologies. Where bandwidth to remote sites is limited, P4P, the Helix Proxy, improves performance by mediating between Perforce applications and the versioning service to cache frequently transmitted file revisions. By intercepting requests for cached file revisions, P4P reduces demand on the Perforce service and the network over which it runs.

To improve performance obtained by multiple Perforce users accessing a shared Perforce repository across a WAN, configure P4P on the side of the network close to the users and configure the users to access the service through P4P; then configure P4P to access the master Perforce service. (On a LAN, you can also obtain performance improvements by setting up proxies to divert workload from the master server's CPU and disks.)

The following diagram illustrates a typical P4P configuration:



In this configuration, file revisions requested by users at a remote development site are fetched first from a central Helix Core server (`p4d` running on `central`) and transferred over a relatively slow WAN. Subsequent requests for that same revision, however, are delivered from the Helix Proxy, (`p4p` running on `outpost`), over the remote development site's LAN, reducing both network traffic across the WAN and CPU load on the central server.

System requirements

To use Helix Proxy, you must have:

- A Helix Versioning Engine at Release 2002.2 or higher (2012.1 or higher 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 you might also want to do the following:

- Enable SSL support. See "Enabling SSL support" on page 101 for more information.
- Defend against man-in-the-middle attacks. See "Defending from man-in-the-middle attacks" on page 101 for more information.

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 Perforce applications.
4. Select the target Helix Core server (**P4TARGET**) for which this proxy will cache.

Windows

Install P4P from the Windows installer's custom/administrator installation dialog.

Running P4P

To run P4P, 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 Core 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, Perforce 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`.

The Perforce proxy supports connectivity over IPv6 networks as well as IPv4. See the *Helix Versioning Engine 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 Versioning Engine via SSL at `ssl:mainserver:1666`.

P4P options

The following command-line options specific to the proxy are supported.

Proxy options:

Option	Meaning
`-d`	Run as daemon - fork first, then run (UNIX only).
`-f`	Do not fork - run as a single-threaded server (UNIX only).
`-i`	Run for `inetd` (socket on `stdin/stdout` - UNIX only).
`-q`	Run quietly; suppress startup messages.
`-c`	Do not compress data stream between the Helix Core 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.

Option	Meaning
`-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 Perforce applications. Default is `P4PORT`, or 1666 if `P4PORT` is not set.
`-t` *port*	Specify the port of the target Helix Core server (that is, the Helix Core server for which P4P acts as a proxy). Default is `P4TARGET` or `perforce:1666` if `P4TARGET` is not set.
`-e` *size*	Cache only those files that are larger than *size* bytes. Default is `P4PFSIZE`, or zero (cache all files) if `P4PFSIZE` is not set.
`-u` *serviceuser*	For proxy servers, authenticate as the specified `serviceuser` when communicating with the central server. The service user must have a valid ticket before the proxy will work.
`-v` *level*	Specifies server trace level. Debug messages are stored in the proxy server's log file. Debug messages from `p4p` are not passed through to `p4d`, and debug messages from `p4d` are not passed through to 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

Option	Meaning
`-l -s`	List pending archive transfers, summarized
`-v lbr.stat.interval=`*n*	Set the file status interval, in seconds. If not set, defaults to 10 seconds.
`-v proxy.monitor.level=`*n*	`0`: (default) Monitoring disabled `1`: Proxy monitors file transfers only `2`: Proxy monitors all operations `3`: Proxy monitors all traffic for all operations
`-v proxy.monitor.interval=`*n*	Proxy monitoring interval, in seconds. If not set, defaults to 10 seconds.

`-m1`
`-m2`
`-m3`
	Show currently-active connections and their status.
	Requires `proxy.monitor.level` set equal to or greater than 1. The optional argument specifies the level of detail: `-m1`, `-m2`, or `-m3` show increasing levels of detail corresponding to the `proxy.monitor.level` setting.

Proxy archive cache options:

Option	Meaning
`-v lbr.proxy.case=`*n*	`1`: (default) Proxy folds case; all files with the same name are assumed to be the same file, regardless of case. `2`: Proxy folds case if, and only if, the upstream server is case-insensitive (that is, if the upstream server is on Windows) `3`: Proxy never folds case.

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 Core 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 Versioning Engine. See "Enabling SSL support" on page 32. The users' Perforce 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.

For complete information about enabling SSL for the proxy, see:
http://answers.perforce.com/articles/KB/2596

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 Perforce 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. For more information about this configurable, see the "Configurables" appendix in *P4 Command Reference*.

Localizing P4P

If your Helix Core server has localized error messages (see "Localizing server error messages" in *Helix Versioning Engine 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 Perforce applications are using the proxy

If your Perforce 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 Perforce 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

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

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

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

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

```
list    group    remotedev    192.168.10.0/24              -//...
list    group    remotedev    [2001:db8:16:81::]/48        -//...
write   group    remotedev    proxy-192.168.10.0/24         //...
write   group    remotedev    proxy-[2001:db8:16:81::]/48   //...
list    group    remotedev    proxy-10.0.0.0/8             -//...
list    group    remotedev    proxy-[2001:db8:1008::]/32   -//...
write   group    remotedev    10.0.0.0/8                    //...
write   group    remotedev    proxy-[2001:db8:1008::]/32    //...
```

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

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

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

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

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

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

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; that is, `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.)

lbr.proxy.case	Behavior
`lbr.proxy.case=1` (default)	Proxy folds case; all files with the same name are assumed to be the same file, regardless of case.

lbr.proxy.case	Behavior
`lbr.proxy.case=2`	Proxy folds case if, and only if, the upstream server is case-insensitive (that is, if the upstream server is on Windows)
`lbr.proxy.case=3`	Proxy never folds case.

After any change to `lbr.proxy.case`, you must clear the cache before restarting the proxy.

Maximizing performance improvement

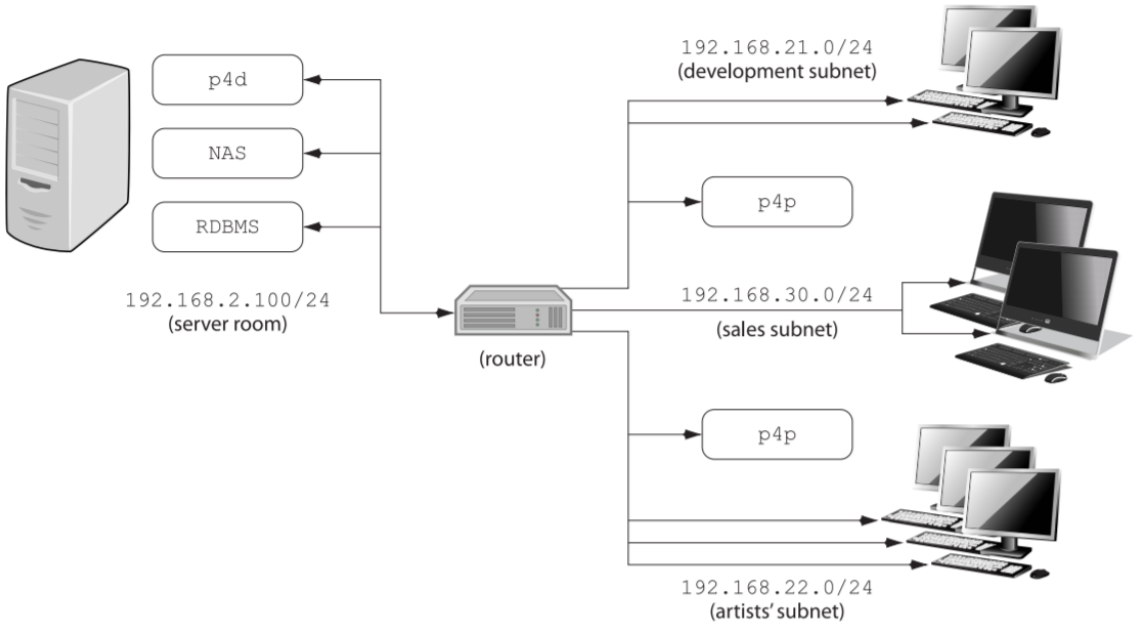
Reducing server CPU usage by disabling file compression

By default, P4P compresses communication between itself and the Perforce 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 Perforce users.

Network topologies versus P4P

If network bandwidth on the same subnet as the Perforce service is nearly saturated, deploying proxy servers on the same subnet will not likely result in a performance improvement. Instead, 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.

For example:



Deploying an additional proxy on a subnet when network bandwidth on the subnet is nearly saturated will not likely result in a performance improvement. Instead, split the subnet into multiple subnets and deploy a proxy in each resulting subnet.

In the illustrated configuration, a server room houses a company's Perforce service (`p4d`), a network storage device (NAS), and a database server (RDBMS). The server room's network segment is saturated by heavy loads placed on it by a sales force constantly querying a database for live updates, and by developers and graphic artists frequently accessing large files versioned by Perforce.

Deploying two instances of the Perforce proxy (one on the developers' subnet, and one on the graphic artists' subnet) enables all three groups to benefit from improved performance due to decreased use on the server room's network segment.

Preloading the cache directory for optimal initial performance

P4P 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 effectively 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 Core server in North America can preload its cache directory by using an automated job that runs a `p4 sync` against the entire Perforce 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 Core 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 Versioning Engine (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 Perforce versioning service. The fourth form of the command is used for system administration tasks involving checkpointing and journaling.

On UNIX and Mac OS X, the executable is **p4d**.

On Windows, the executable is **p4d.exe** (running as a server) or **p4s.exe** (running as a service).

Exit Status

After successful startup, **p4d** does not normally exit. It merely outputs the following startup message:

```
Perforce server starting...
```

and runs in the background.

On failed startup, **p4d** returns a nonzero error code.

Also, if invoked with any of the **-j** checkpointing or journaling options, **p4d** exits with a nonzero error code if any error occurs.

Options

Server options	Meaning
-d	Run as a daemon (in the background)
-f	Run as a single-threaded (non-forking) process

Server options	Meaning
`-i`	Run from `inetd` on UNIX
`-q`	Run quietly (no startup messages)
`--pid-file [=file]`	Write the PID of the server to a file named `server.pid` in the directory specified by `P4ROOT`, or write the PID to the file specified by *`file`*. This makes it easier to identify a server instance among many. The *`file`* parameter can be a complete path specification. The file does not have to reside in `P4ROOT`.
`-xi`	Irreversibly reconfigure the Helix Versioning Engine (and its metadata) to operate in Unicode mode. Do not use this option unless you know you require Unicode mode. See the *Release Notes* and *Internationalization Notes* for details.
`-xu`	Run database upgrades and exit. This will no longer run automatically if there are fewer than 1000 changelists. Upgrades must be run manually unless the server is a DVCS personal server; in this case, any upgrade steps are run automatically.
`-xv`	Run low-level database validation and quit.
`-xvU`	Run fast verification; do not lock database tables, and verify only that the unlock count for each table is zero.
`-xD [ serverID ]`	Display (or set) the server's *`serverID`* (stored in the `server.id` file) and exit.

General options	Meaning
`-h`, `-?`	Print help message.
`-V`	Print version number.
`-A auditlog`	Specify an audit log file. Overrides `P4AUDIT` setting. Default is null.
`-Id description`	A server description for use with `p4 server`. Overrides `P4DESCRIPTION` setting.
`-In name`	A server name for use with `p4 configure`. Overrides `P4NAME` setting.
`-J journal`	Specify a journal file. Overrides `P4JOURNAL` setting. Default is `journal`. (Use `-J off` to disable journaling.)
`-L log`	Specify a log file. Overrides `P4LOG` setting. Default is `STDERR`.

General options	Meaning
`-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	Meaning
`-c command`	Lock database tables, run *command*, unlock the tables, and exit.
`-jc [ prefix ]`	Journal-create; checkpoint and `.md5` file, and save/truncate journal. In this case, your checkpoint and journal files are named *prefix*`.ckp.`*n* and *prefix*`.jnl.`*n* respectively, where *prefix* is as specified on the command line and *n* is a sequence number. If no *prefix* is specified, the default filenames `checkpoint.`*n* and `journal.`*n* are used. You can store checkpoints and journals in the directory of your choice by specifying the directory as part of the prefix. **Warning** If you use this option, it must be the last option on the command line.
`-jd file`	Journal-checkpoint; create checkpoint and `.md5` file without saving/truncating journal.
`-jj [ prefix ]`	Journal-only; save and truncate journal without checkpointing.
`-jr file`	Journal-restore; restore metadata from a checkpoint and/or journal file. If you specify the `-r $P4ROOT` option on the command line, the `-r` option must precede the `-jr` option.

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's zipped can it be successfully unzipped? ■ If it has an MD5 file with its MD5, does it match? ■ Does it have the expected header and trailer? This command does not replay the journal. Use the `-z` option with the `-jv` option to verify the integrity of compressed journals or compressed checkpoints.
`-z`	Compress (in `gzip` format) checkpoints and journals. When you use this option with the `-jd` option, Perforce automatically adds the `.gz` extension to the checkpoint file. So, the command: `p4d -jd -z myCheckpoint` creates two files: `myCheckpoint.gz` and `myCheckpoint.md5`.
`-Z`	Compress (in `gzip` format) checkpoint, but leave journal uncompressed for use by replica servers. That is, it applies to `-jc`, not `-jd`.

Journal restore options	Meaning
`-jrc file`	Journal-restore with integrity-checking. Because this option locks the database, this option is intended only for use by replica servers started with the `p4 replicate` command.
`-jrF file`	Allow replaying a checkpoint over an existing database. (Bypass the check done by the `-jr` option to see if a checkpoint is being replayed into an existing database directory by mistake.)
`-b bunch -jr file`	Read *bunch* lines of journal records, sorting and removing duplicates before updating the database. The default is `5000`, but can be set to `1` to force serial processing. This combination of options is intended for use with by replica servers started with the `p4 replicate` command.
`-f -jr file`	Ignore failures to delete records; this meaning of `-f` applies only when `-jr` is present. This combination of options is intended for use with by replica servers started with the `p4 replicate` command. By default, journal restoration halts if record deletion fails. As with all journal-restore commands, if you specify the `-r $P4ROOT` option on the command line, the `-r` option must precede the `-jr` option.

Journal restore options	Meaning
`-m -jr file`	Schedule new revisions for replica network transfer. Required only in environments that use `p4 pull -u` for archived files, but `p4 replicate` for metadata. Not required in replicated environments based solely on `p4 pull`.
`-s -jr file`	Record restored journal records into regular journal, so that the records can be propagated from the server's journal to any replicas downstream of the server. This combination of options is intended for use in conjunction with Perforce technical support.

Replication and multi-server options	Meaning
`-a host:port`	In multi-server environments, specify an authentication server for licensing and protections data. Overrides `P4AUTH` setting. Default is null.
`-g host:port`	In multi-server environments, specify a changelist server from which to obtain changelist numbers. Overrides `P4CHANGE` setting. Default is null.
`-t host:port`	For replicas, specify the target (master) server from which to pull data. Overrides `P4TARGET` setting. Default is null.
`-u serviceuser`	For replicas, authenticate as the specified *serviceuser* when communicating with the master. The service user must have a valid ticket before replica operations will succeed.

Journal dump/restore filtering	Meaning
`-jd file db.table`	Dump `db.table` by creating a checkpoint *file* that contains only the data stored in `db.table` This command can also be used with non-journaled tables.
`-k db.table1,db.table2,... -jd file`	Dump a set of named tables to a single dump *file*.

Journal dump/restore filtering	Meaning
`-K db.table1,db.table2,... -jd file`	Dump all tables except the named tables to the dump *file*.
`-P serverid -jd file`	Specify filter patterns for `p4d -jd` by specifying a *serverid* from which to read filters (see `p4 help server`, or use the older syntax described in `p4 help export`.) This option is useful for seeding a filtered replica.
`-k db.table1,db.table2,... -jr file`	Restore from *file*, including only journal records for the tables named in the list specified by the `-k` option.
`-K db.table1,db.table2,... -jr file`	Restore from *file*, excluding all journal records for the tables named in the list specified by the `-K` option.

Certificate Handling	Meaning
`-Gc`	Generate SSL credentials files for the server: create a private key and certificate file in `P4SSLDIR`, and then exit. Requires that `P4SSLDIR` be set to a directory that is owned by the user invoking the command, and that is readable only by that user. If `config.txt` is present in `P4SSLDIR`, generate a self-signed certificate with specified characteristics.
`-Gf`	Display the fingerprint of the server's public key, and exit. Administrators can communicate this fingerprint to end users, who can then use the `p4 trust` command to determine whether or not the fingerprint (of the server to which they happen to be connecting) is accurate.

Configuration options	Meaning
`-cshow`	Display the contents of `db.config` without starting the service. (That is, run `p4 configure show allservers`, but without a running service.)

`-cset` *`server`* `#var=val`	Set a Perforce configurable without starting the service, optionally specifying the server for which the configurable is to apply. For example,
	`p4d -r . "-cset replica#P4JOURNAL=off"`
	`p4d -r . "-cset replica#P4JOURNAL=off replica#server=3"`
	It is best to include the entire *`variable=value`* expression in quotation marks.

`-cunset` *`server`*`#var`	Unset the specified configurable.

Usage Notes

- On all systems, journaling is enabled by default. If `P4JOURNAL` is unset when `p4d` starts, the default location for the journal is `$P4ROOT`. If you want to manually disable journaling, you must explicitly set `P4JOURNAL` to `off`.
- Take checkpoints and truncate the journal often, preferably as part of your nightly backup process.
- Checkpointing and journaling preserve only your Perforce metadata (data *about* your stored files). The stored files themselves (the files containing your source code) reside under `P4ROOT` and must be also be backed up as part of your regular backup procedure.
- It is best to keep journal files and checkpoints on a different hard drive or network location than the Perforce database.
- If your users use triggers, don't use the `-f` (non-forking mode) option; the Perforce service needs to be able to spawn copies of itself ("fork") in order to run trigger scripts.
- After a hardware failure, the options required for restoring your metadata from your checkpoint and journal files can vary, depending on whether data was corrupted.
- Because restorations from backups involving loss of files under `P4ROOT` often require the journal file, we strongly recommend that the journal file reside on a separate filesystem from `P4ROOT`. This way, in the event of corruption of the filesystem containing `P4ROOT`, the journal is likely to remain accessible.
- The database upgrade option (`-xu`) can require considerable disk space. See the *Release Notes* for details when upgrading.

Related Commands

To start the service, listening to port `1999`, with journaling enabled and written to `journalfile`.	`p4d -d -p 1999 -J /opt/p4d/journalfile`

To checkpoint a server with a non-default journal file, the `-J` option (or the environment variable `P4JOURNAL`) must match the journal file specified when the server was started.	Checkpoint with:
	`p4d -J /p4d/jfile -jc`
	or
	`P4JOURNAL=/p4d/jfile ; export P4JOURNAL; p4d -jc`

To create a compressed checkpoint from a server with files in directory `P4ROOT`	`p4d -r $P4ROOT -z -jc`
To create a compressed checkpoint with a user-specified prefix of “ckp” from a server with files in directory `P4ROOT`	`p4d -r $P4ROOT -z -jc ckp`
To restore metadata from a checkpoint named `checkpoint.3` for a server with root directory `P4ROOT`	`p4d -r $P4ROOT -jr checkpoint.3` (The `-r` option must precede the `-jr` option.)
To restore metadata from a compressed checkpoint named `checkpoint.3.gz` for a server with root directory `P4ROOT`	`p4d -r $P4ROOT -z -jr checkpoint.3.gz` (The `-r` option must precede the `-jr` option.)

Helix Versioning Engine Control (p4dctl)

The Perforce Service Control (`p4dctl`) utility allows you to manage Perforce services running on the local host. Non-root users can administer the services they own, while `root` may administer all services, but may not own any.

> **Note**
> `p4dctl` can only be obtained as part of the UNIX package installation. It is not supported on Windows.

You use the `p4dctl` utility to configure the environment in which services run and to manage the services themselves. The basic workflow for an administrator using the `p4dctl` utility is as follows:

1. Edit a configuration file that defines the environment for the services you want to control.
2. Execute `p4dctl` commands to start and stop services, to get information about services, and to checkpoint services.

You can use a single `p4dctl` command to manage all services or an arbitrary group of services by assigning them a common *name* in the `p4dctl` configuration file.

`p4dctl` introduces no new environment variables; it enforces strict control of the environment of any service it starts according to the directives in the `p4dctl` configuration file. This prevents failures that stem from the differences between the user's environment and that of `root`.

Installation

`p4dctl` is installed as part of the UNIX package installation. The installation process automatically creates a master configuration file located at `/etc/perforce/p4dctl.conf`.

As part of the package install, `p4dctl` is installed as a `setuid` root executable because it uses root privileges to maintain pid files for compatibility with systems that use them. For all other operations, `p4dctl` runs with the privileges of the executing user. This allows non-root users to start and stop the services they own while having the pid file remain up to date.

Configuration file format

`p4dctl` uses a configuration file, `p4dctl.conf`, to control the following:

- service settings for the services started with the `p4dctl` command.
- settings for the `p4dctl` utility itself
- service processes managed by `p4dctl`, for example checkpointing and journal rotation

- the environment in which managed services are running

  The environment is configured using environment variables that may be defined globally or for a specific service. The service type determines which variables must be defined. See "Service types and required settings" on page 120 for information on the requirements for each type.

A `p4dctl` configuration file is made up of an `environment` block and one or more `server` type blocks. The following sections describe each type in detail.

The configuration file may also contain comments; these are designated by starting the comment line with the `#` sign.

Settings specified outside of a server block are global and are merged into the settings of all services. They take the following form:

setting_name = *value*

For example:

```
PATH = /bin:/user/bin
```

Environment block

An environment block defines environment variables that are applied to one or more services. You can have more than one environment block. Server-specific environment blocks settings override corresponding settings in global environment blocks.

An environment block is defined using the following syntax:

```
Environment
{
   variable = value
}
```

An environment block may be used outside of a server block or inside of it.

- If the block is outside a server block, the variables it contains are applied to the environment of all processes created by `p4dctl`.
- If the block is inside a server block, the variables it defines are set only in the environment of that server's processes, but they do override corresponding settings at the environment level.

For example, the following settings outside a server block ensure that the owner is set to `perforce`, logging is enabled, and the correct `P4CONFIG` files are used.

```
Environment
{
     P4DEBUG     =  "server=1" # Embedded = requires quotes
     P4LOG       =  log
```

```
    P4CONFIG    =   .p4config
}
```

Server block

A server block defines settings and variables that apply only to the specified type of service. Type may be one of the following:

Type	Meaning
`p4d`	Helix Core server
`p4p`	Perforce proxy server
`p4broker`	Perforce broker
`p4ftp`	Perforce FTP plugin
`p4web`	Perforce web client
`other`	Any other service

A server block is defined using the following syntax:

```
server_type name
{
   setting = value
   Environment
   {
       variable = value
   }
}
```

The specified `name` name must refer to services of a given type, but the name can include different types of servers. This allows you to control or query groups of heterogeneous servers that share the same name.

For example, a configuration that defines p4d, proxy, and p4ftp services all using the name `main` can use a command like the following to stop all these services without affecting any other services.

```
$ p4dctl stop main
```

You can define the following variables within server blocks. `Owner` and `Execute` are required for all server types.

Setting	Meaning
`Owner`	The owner of the service. The service is started under the owner's account and with their privileges. The user can also use `p4dctl` to manage the server they own. Required.
`Execute`	The path to the binary to execute when starting this server. Required.
`Args`	A string containing the arguments to be passed to the binary specified with `Execute`. The string *must* be quoted if it contains a space.
`Enabled`	Set to `FALSE` to disable the service and not start it with the `p4dctl start` command. Default: `TRUE`
`Umask`	An octal value specifying the `umask` to be applied to the child processes for this service. The default `umask` on most Linux/Unix systems is 022, which means all new files are readable by all users. Setting this variable to 077 ensures that the files created by this service are only accessible to the owner of the service.
`Prefix`	A string containing a prefix to apply when checkpointing the server or rotating the journal. This prefix is passed down to the relevant `p4d` command if needed. Default: none

Setting	Meaning

`PrettyNames`	Set to `true` to have `p4dctl` format the names of the server processes it starts, in an informative way.
	In the following example, the `p4d` process is qualified with its host and port name when `PrettyNames` is set to true.
	`PrettyNames=true`
	`  perforce callto:21397%201%200%2010[21397 1  0 10]:48 ?`
	`00:00:00 p4d`
	
	`[blacksphere/1666]`
	`PrettyNames=false`
	`  perforce callto:21725%201%200%2010[21725 1  0 10]:50 ?`
	`00:00:00`
	
	`/usr/sbin/p4d`
	Default: true

Service types and required settings

Each service type requires that you define the `owner` of the server (which cannot be `root`) and the `execute` path where its binary can be found. For example, for the `p4d` type, you specify the path to the `p4d` binary, for the broker, you must provide the path to the `p4broker` binary, and so on.

For each service type, you must also define certain environment variables; these are listed in the following subsections.

Type	Variable	Setting
`p4d`	`P4PORT`	Port to use for this service
	`P4ROOT`	Path to the server's root directory
	`PATH`	Search path to be used for this service
`p4p`	`PORT`	Port to use for this service
	`P4TARGET`	Address of the target Perforce service
	`P4ROOT`	Path to the server's root directory
	`PATH`	Search path to be used for this service
`p4broker`	`P4BROKEROPTIONS`	Command line options to pass to this broker

Type	Variable	Setting
p4ftp	PORT	Address of the target Perforce service
	P4FTPPORT	Port to use for serving FTP requests
p4web	PORT	Address of the target Helix Core server
	P4WEBPORT	Port to use for serving HTTP requests
	P4ROOT	Path to the server's root directory
	PATH	Search path to be used for this service

Configuration file examples

The following example shows a basic Helix Core server (p4d) configuration file.

```
p4d minimum
{
  Owner   = perforce
  Execute = /usr/bin/p4d
  Environment

  {
   P4ROOT     = /home/perforce/p4-main
   P4PORT     = 1666
   PATH       = /bin:/usr/bin:/usr/local/bin
  }
}
```

In the following example, the PATH environment variable is defined once, globally for both the service and its proxy. Note how the name 'test' is used to refer to both.

```
Environment
{
  PATH       = /bin:/usr/bin:/usr/local/bin
}

p4d test
{
  Owner   = perforce
  Execute = /usr/bin/p4d
```

```
Environment
  {
   P4ROOT     = /home/perforce/p4-main
   P4PORT     = "localhost:1667"
  }
}

p4p test
{
  Owner   = perforce
  Execute = /usr/bin/p4p

  Environment
  {
   P4ROOT     = /home/perforce/proxy-main
   P4PORT     = 1666
   P4TARGET   = "localhost:1667"
  }
}
```

Using multiple configuration files

You can modularize your configuration by creating multiple configuration files and directories and including these in your configuration.

- To include a specific file, use the following syntax:

  ```
  include pathToFile
  ```

- To include directories, use the following syntax:

  ```
  include directoryPath
  ```

  When including directories, **p4dctl** requires that names for files included end in **.conf**.

The following example shows a multiple file configuration.

```
Environment
{
  PATH       = /bin:/usr/bin:/usr/local/bin
}
```

```
include /etc/perforce/p4dctl.conf.d
```

p4dctl commands

`p4dctl` commands can be divided into three categories: commands that stop and start services, commands that checkpoint services, and commands that return information about services.

The `p4dctl checkpoint` command is similar to the `p4d -jc` command.

The following table presents a summary of command syntax for each category. The parameter `-a` specifies all servers.

Category	Syntax

Control services	`p4dctl [ options ] start [ -t type ] -a`
	`p4dctl [ options ] start [ -t type ] name`
	`p4dctl [ options ] stop [ -t type ] -a`
	`p4dctl [ options ] stop [ -t type ] name`
	`p4dctl [ options ] restart [ -t type ] -a`
	`p4dctl [ options ] restart [ -t type ] name`

Checkpoints and journals	`p4dctl [ options ] checkpoint -a` `p4dctl [ options ] checkpoint name`

Query services	`p4dctl [ options ] status [ -t type ] -a`
	`p4dctl [ options ] status [ -t type ] name`
	`p4dctl [ options ] list [ -t type ]`
	`p4dctl [ options ] list [ -t type ] name`
	`p4dctl [ options ] env [ -t type ] -a var [var…]`
	`p4dctl [ options ] status [ -t type ] name var [var…]`

Options to `p4dctl` commands are described in the following table. The meaning of variable names other than option names is explained in "Configuration file format" on page 116.

Options	Meaning
`-c configFile`	Path to the configuration file Default: `/etc/perforce/p4dctl.conf`
`-p pidDir`	Path to the pid file directory. Default: `/var/run`
`-q`	Send output to syslog instead of `STDOUT` or `STDERR`

Options	Meaning
`-v` *level*	Set debug level (0-9) For more information, see the description of the **P4DEBUG** environment variable in *P4 Command Reference*.
`-V`	Display version and exit.

License Statements

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

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

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

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