HelixCore

Solutions Overview: Helix Version Control System

2017.2

October 2017

PERFORCE

www.perforce.com

Contents

How to use this guide

This manual introduces Helix Server, a secure, scalable, and highly available version control system that supports parallel development. You should read this document before you start working with Helix Server.

This document:

- introduces the basic concepts and tasks of version control
- explains how you can configure Helix Server to improve performance and to scale the system
- suggests some ways you can extend and customize Helix Server
- explains how you can use Helix Server with other products to get additional functionality
- discusses use cases for Helix Server
- introduces resources that can help you use Helix Server

If you are familiar with version control systems, you can skip ahead to "Helix Server as a version control implementation" on page 8.

Feedback

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

Other documentation

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

Syntax conventions

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

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
[`-f`]	The enclosed elements are optional. Omit the brackets when you compose the command.

Notation	Meaning
...	■ 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.

The basics of version control

When you work alone on a document, the latest is usually the greatest: you successively open the document, make changes, and save the document. Each time you save, you overwrite the existing copy. The situation is different when you are working with a large, globally distributed team on a project consisting of hundreds, or even thousands, of files. In this case, it is important to track authorship and changes, and to resolve conflicts when users make conflicting changes to the same file. Version control systems allow you to do this. You can track and manage changes to any large collection of digital assets: documents, source code, web sites, audio files, and so on.

One technique of version control systems is versioning. Rather than overwriting earlier versions of a file when it is saved, each saved copy of the file is versioned and assigned a number or letter that reflects the order in which it was saved.

In addition to identifying a file version within a sequence of versions, a version control system automatically associates certain information with each version: it records who made the change, when the change was made, and why the change was made. This information provides an audit trail that you can always consult to understand how a project developed and when specific changes were made. Because no version of a file is overwritten, when bugs arise, it is possible to identify the point at which the bug was introduced. This can be critical in fixing bugs that cannot be reproduced. In addition, looking at file history and understanding why certain decisions were made can help project participants stay on track or find appropriate options for future directions.

Sharing data under version control requires a certain amount of gatekeeping to determine who can access the data and how conflicts are resolved when two users make changes to the same file. To support this gatekeeping function, version control systems introduce the additional step of checking out a file and checking in or submitting a file.

The basic version control workflow looks like this:

1. Assets under version control are placed in a specified repository.
2. Assets are associated with specific permissions that enable users to read or modify them.
3. A user checks out a working copy of an asset and makes changes.
4. Another user checks out a working copy of the same asset and makes changes.
5. The first user saves changes to the local working copy and checks in that copy.
6. The second user saves changes to the local working copy and attempts to check in that copy.
7. The version control system detects the fact that the same asset was changed in parallel, and it asks the second user to merge changes with those of the first user before the second user's changes can be checked in. The work of comparing and merging changes is called *resolving*.

In this way, the version control system makes sure that changes are predictable, manageable, and auditable.

Version control systems are traditionally either centralized or distributed:

- *Centralized version control systems* use a single repository from which users check out one or more files to work on locally.

- *Distributed version control systems* allow users to host repositories locally, check out entire repositories with all history—or, in the case of Helix Server, a subset of repositories—work independently of one another, and combine their work through merging when necessary.

Helix Server supports either model, as well as a hybrid of the two.

Version control systems can be used as stand-alone applications or they can be incorporated into development or authoring tools as a means of managing the assets produced by these tools.

Helix Server as a version control implementation

Helix Server uses a client-server architecture to implement version control management.

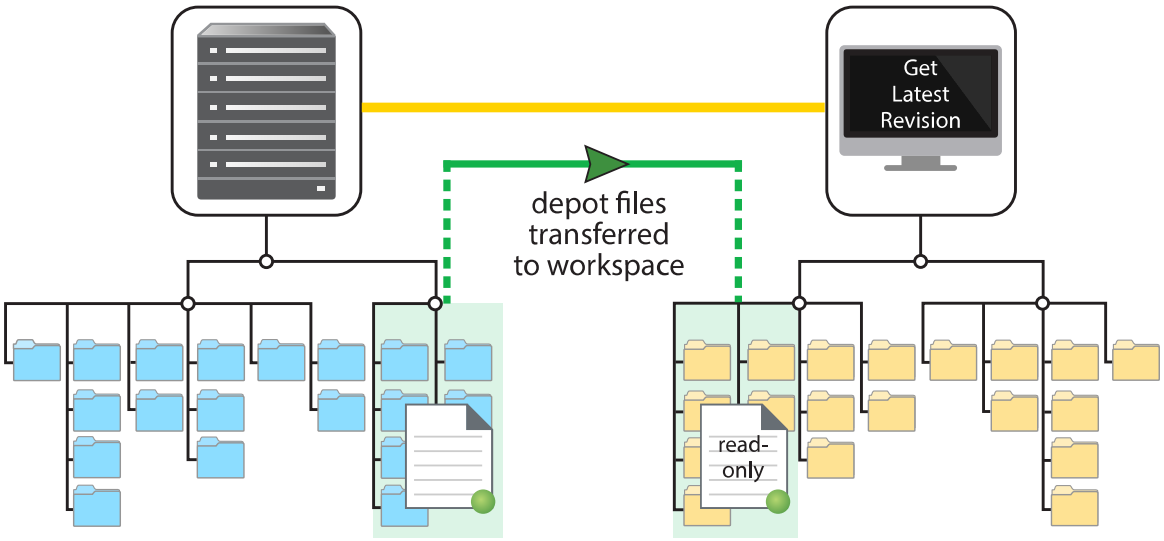
- The Helix Server (also known as the Helix Versioning Engine or `p4d`) manages shared file repositories, or depots, that contain every revision of every file under version management. Files are organized into directory trees. The server also maintains a database to track data associated with files and client activity: logs, user permissions, metadata, configuration values, and so on.
- Helix Server *clients* provide an interface that allows you to check files in and out of the depot, resolve conflicts, track change requests, and more.

  Helix Server clients include a command-line client, a graphical user interface client, and various plugins that work with commercial IDEs and productivity software. A Helix Server server can provide services to a mix of Helix Server clients.

  You also use Helix Server clients to manage a special area of your computer called a workspace. Directories in the depot are mapped to directories in your workspace, which contain local copies of managed files. You always work on managed files in your workspace:

  1. You check the files out of the depot (and into your workspace).
  2. You modify the files.
  3. You check them back into the depot.
  4. If the changes you try to submit conflict with changes that other users, working in parallel with you, have already submitted, you must resolve conflicts as needed.

The next figure shows the mapping between depot files (shown on the left) and workspace files (shown on the right). Until files are checked out from the depot, they remain as read-only in the workspace. To have Helix Server update your workspace so that it reflects current work on the depot, synchronize your workspace to the depot by getting the latest revision of the files.



We have explained about checking files in and out of the depot, suggesting that single files may be checked in and out. In fact, the means we use to check files in and out of the depot is the *changelist*. A changelist must contain at least one file and may contain tens of thousands. A changelist is numbered and allows you to track all changes with respect to the contents of the depot: file modifications, the addition of a file, or the deletion of a file.

The changelist is the simplest way to organize your work. A changelists also represents the atomic unit of work in Helix Server: if a changelist includes several files, changes for all the files are committed to the depot or none of the changes are. For example, if a network connection between the client and the server fails during changelist submission, the entire submit fails.

Multiple user access to a set of files

Version control systems must address the fundamental need for multiple users to work on the same project simultaneously. Helix Server offers two ways to do this:

- File locking: Helix Server locks a file while someone is working in it. This controls access to the file: if several users want to edit the same file, it is possible to merge changes into one mutually acceptable version.
- Branching and merging: By branching streams and then merging them later, multiple users can work on the same files simultaneously. See "Balancing stability and innovation: the mainline model" below.

Balancing stability and innovation: the mainline model

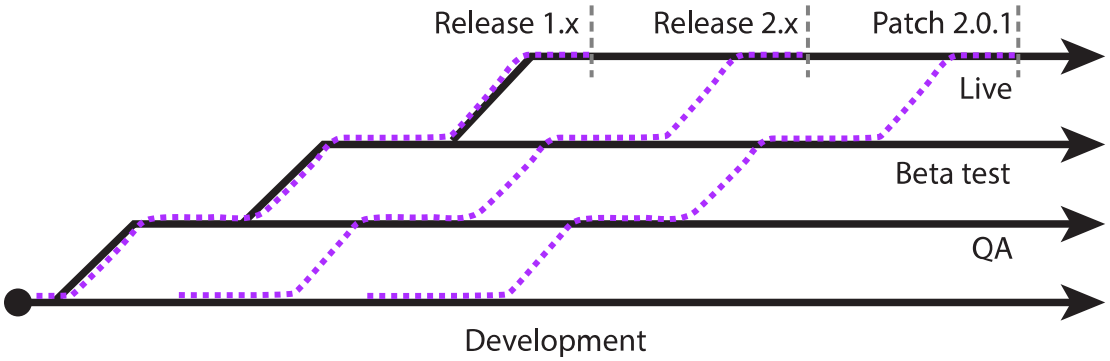
So far, we have explained how version control systems handle the problem of different users working on the same file at the same time: when the different copies of the file are checked in, all but the first user must resolve changes by deciding which changes will be preserved in the latest version of the file.

This is the simplest use case for parallel development. More complicated cases arise when large development projects require many people to work in parallel both because they must support multiple releases and because they involve multiple functional teams working together to create the desired product.

For example, a game development company depends on the combined efforts of artists, musicians, programmers, testers, and build engineers to create a release. One way to organize this effort is to split the set of files that make up the project into multiple parallel branches, allowing development and testing to occur along each branch. Integration then occurs across branches to promote everyone's work into a releasable product.

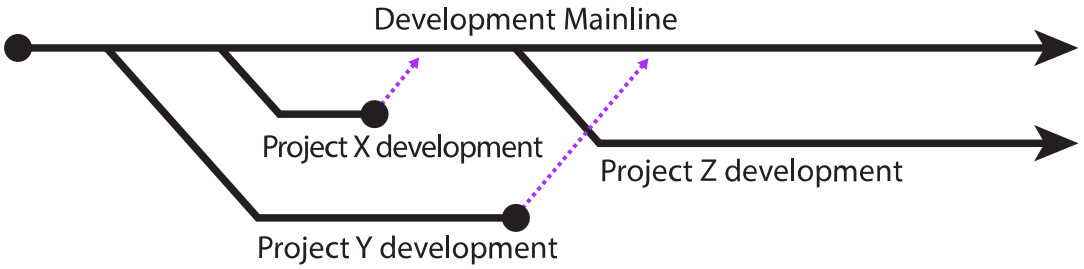
Consider the following cases:

- Extremely short release cycles, such as occur with fast changing web content, require overlapping cycles of testing and release.



  To handle this case, we move code through the branches shown above. Development occurs along the development line. When a milestone is reached, code is copied up to the QA line where it is thoroughly tested. After it clears all tests, it is copied up to the Beta test line where it is subjected to real-world use. Having satisfied Beta users, it can then be copied up to a release line. Note the purple dashed lines, which indicate the flow of code as it is copied up through the branch hierarchy.

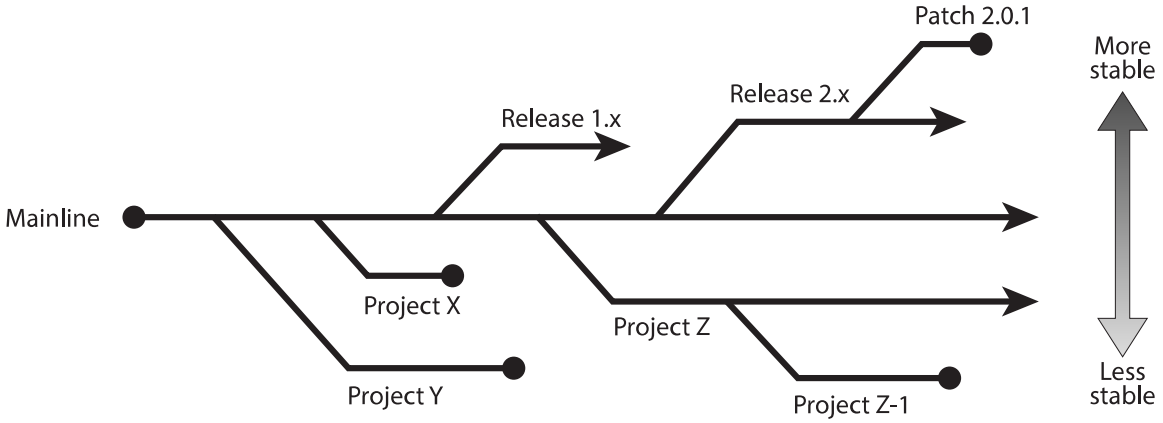
- Unforeseen delays cause some features under development to miss deadlines. Not wanting to imperil the project as a whole, the project is released while development continues for the laggard components.



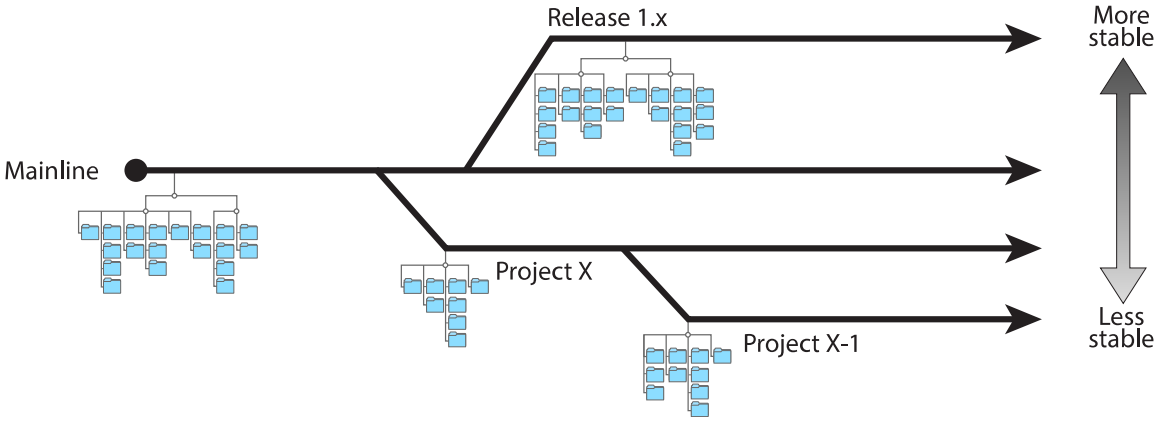
  In the model shown above, projects X, Y, and Z have been branched off from the development mainline and worked on independently. When work has completed and projects X and Y have passed development tests, they are copied up to the mainline. Project Z however could not be completed and work continues on that project without putting the larger release at risk.

What does branching have to offer? It allows us to balance the need for stability with the need for innovation. On the one hand, we have release branches that hold the most tested and stable code; on the other hand, we have development branches that allow for experimentation and exploration without putting the release at risk.

Branches can be organized in a variety of ways: you can create branches for different platforms, you can create branches along organizational lines, and so on. One common model used for product development is the Mainline model shown below:



The Mainline forms the trunk from which release branches and development branches are created. Each branch normally contains a subset of the Mainline files. Release branches might contain fewer files because files needed for testing are excluded; development branches might contain a different subset of files because the projects they represent focus on discrete product features. In the Mainline model, the "up" direction indicates increased stability or confidence.

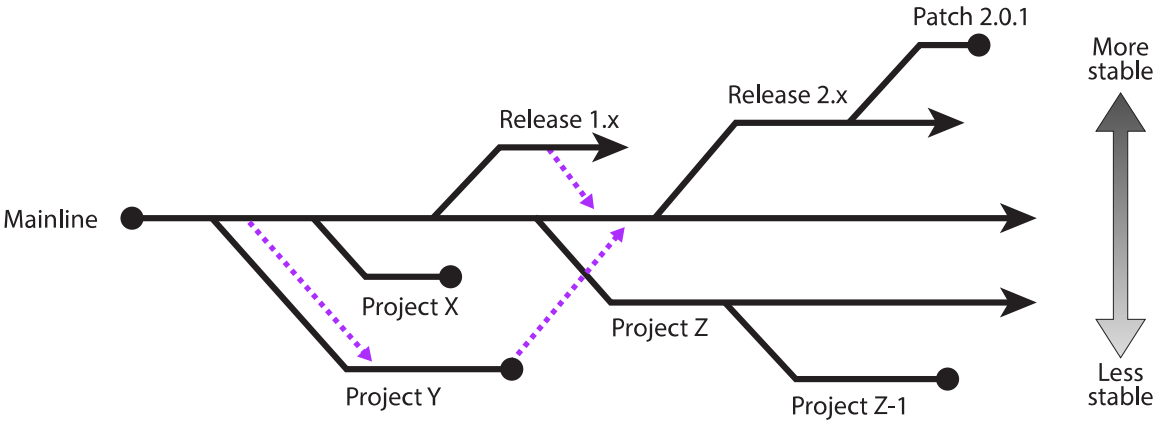


When you create branches, you are free to integrate changes in any direction you like. Unfortunately, this can lead to big problems if you inadvertently integrate untested changes into an otherwise stable branch. For this reason, in addition to defining branches that isolate changes, the Mainline model is most useful when it can implement some protocols that limit what changes can be made and in what direction.

Streams

Helix Server streams implement the Mainline model, adding intelligence that determines what changes can be made and in what order they must be made.

Let's look at the Mainline example again and add some information to indicate flow of change:

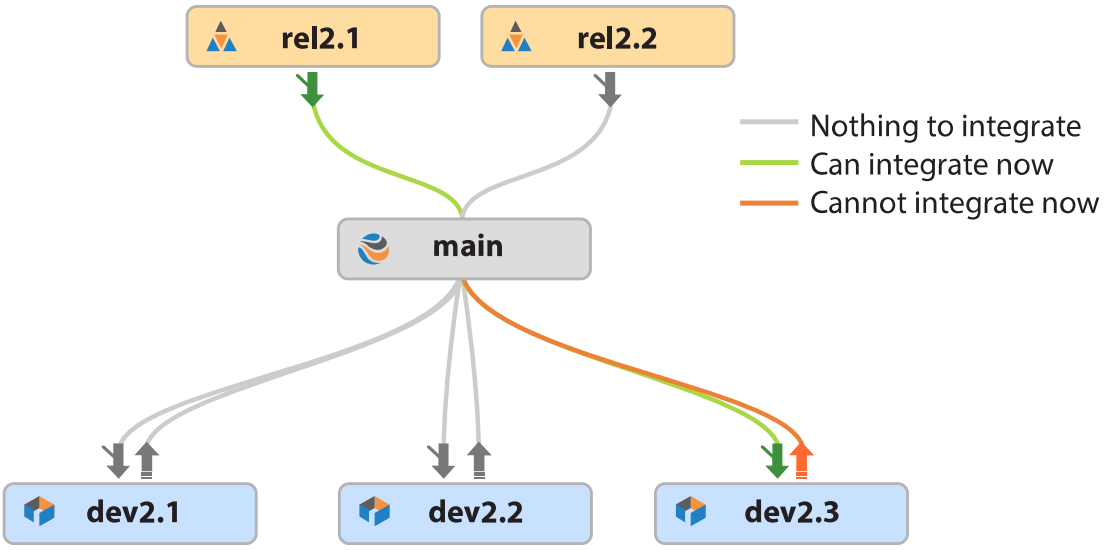


- Project Y has been branched from Mainline; work and testing continues until it is complete. It is then ready to be copied back up to the mainline. However, while development has taken place in Y, Mainline has continued to change. Before we can copy the contents up to the Mainline, we need to make sure that Project Y files reflect changes that have been made in Mainline; we merge those changes into Project Y before we copy Project Y files to Mainline.
- A bug is found in Release 1.x. The bug is fixed and tested. We now want the bug fix in Mainline, so we merge files from Release 1.x down to Mainline. We do not copy anything up because Release 1.x should not include any features added after it has been branched.

The Mainline model arranges branches in terms of stability: the most stable branches are at the top; the least stable branches are at the bottom. The flow of change needs to support this model by merging changes down and copying up.

Typically, when you work with streams, you define and populate the mainline first. You then create development streams and release streams as children of the Mainline stream. The type of a stream and its relationship to other streams determines what sort of changes can be made and in what order they are made.

Rather than using a timeline, the streams GUI—found in the Helix Visual Client (P4V)--represents related streams as shown below:



The children of Main are shown both above and below Main. Release type streams are at the top; development and task streams are at the bottom. Stability grows as streams near the top of the diagram. The direction and color of arrows linking streams indicate both the direction of flow and the order of flow.

When you create a stream, you specify its type, its relationship to other streams, and how files are to be treated for merging and branching. The information you provide is then used by the streams application to encourage good behavior.

Streams provide visual clues for where and how to branch and merge. They guide behavior that supports stability and innovation. Using streams eliminates much of the work needed to define branches, to create workspaces, and to manage integrations.

An additional advantage of using streams is that when you switch from one stream to another, the contents of your workspace are updated automatically to reflect the contents of the current stream.

Streams automate branching, but you do not have to use them. You can create your own branches and manage them as you see fit. Custom branching gives you finer grained control but you lose the convenience of built-in flow control and workspace updating.

For information on streams, see the *Helix Versioning Engine User Guide*.

Organizing your work: jobs and labels

In addition to using changelists and streams to organize your work, you can use two other methods: jobs and labels.

- *Jobs* provide lightweight issue tracking that integrates well with third party defect tracking and workflow systems. They allow you to track the status of a bug or an enhancement request. Jobs have a status and a creator and are associated with changelists that implement the bug fix or the enhancement. An administrator can customize the type of information tracked by jobs add more fine grained status values, or define additional fields for information to be tracked: which customer

the enhancement is for; what was done to test the fix, and so on.

You can integrate the jobs function with third-party defect tracking and workflow systems. For more information, see the *Defect Tracking Gateway* page.

- *Labels* are sets of tagged file revisions that allow you to handle a heterogeneous group of files as one unit. While a changelist refers only to the contents of a given set of files at the time they were submitted, a label can refer to a group of file revisions from different points in time. You might want to use labels to define the group of files contained in a particular release, to sync a set of files, to populate a workspace, or to specify a set of file revisions to be branched. You can also use a label as an alias for a changelist number, which makes it easier to remember the changelist and easier to refer to it in issuing commands.

For information about jobs and labels from a user's perspective, see the *Helix Versioning Engine User Guide*.

For information about managing jobs and labels, see the *Helix Versioning Engine Administrator Guide: Fundamentals*.

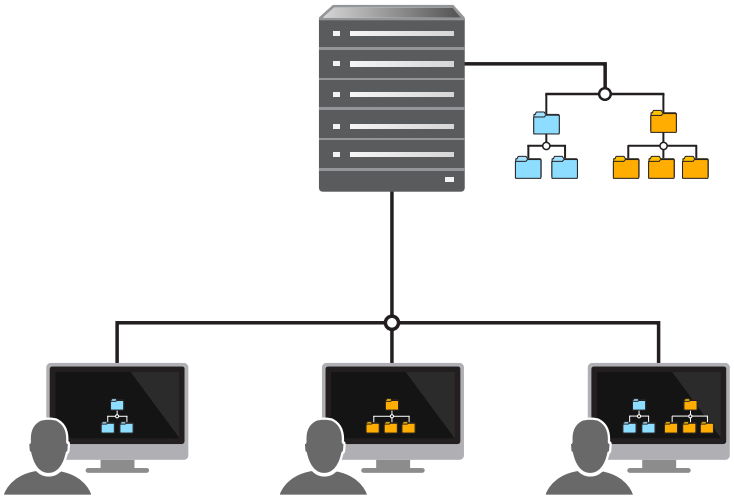
Working together and working apart: centralized and distributed development

We mentioned earlier that version control systems can implement either a centralized model or a distributed model:

- *Centralized version control systems* use a single repository from which users check out one or more files to work on in their local directories.
- *Distributed version control systems* allow users to host repositories locally, check out entire repositories with history—or, in the case of Helix Server, a subset of repositories—work independently of one another, and combine their work through merging when necessary.

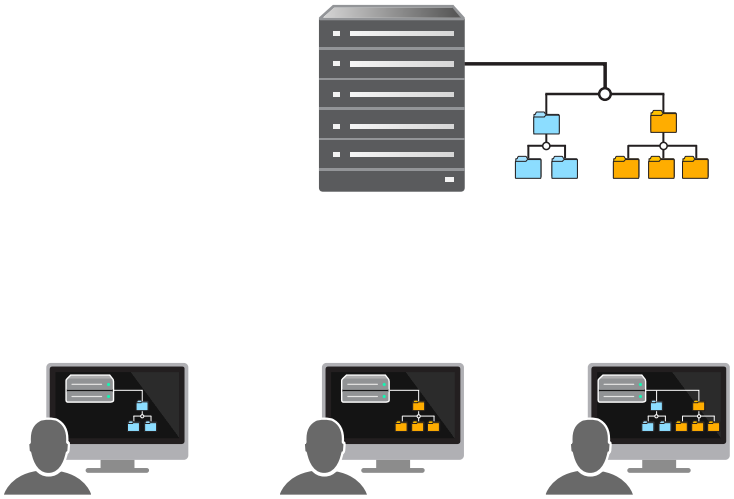
Helix Server supports either model, as well as a hybrid of the two.

In the centralized model, clients work with a depot on a shared server. A mapping of files from the depot to their workspace determines which files they are able to work with in their workspace:



Users check files out of the same depot, work on them, and check in their changed files. If multiple users work on the same file, they use merging and conflict resolution to make sure the resulting version is satisfactory to all authors. Although users can disconnect from the shared server and continue to work on the files in their workspace, some manual work is required to sync back to the server and to check in files when the user reconnects. For information on working with Helix Server using this model, see the *Helix Versioning Engine User Guide*.

In the distributed model, users work with a depot on personal servers that are then connected to a shared server. The depot on their personal server might contain a subset or the entire set of files on the shared server. Each user can work disconnected from the shared server but still be able to access all the files in their workspace and place these under version control using their personal server. Each user can access the entire history of a file locally, rewrite and revise history, and manage the files and streams on their local machine without interacting with the shared server:



When users decide to share their code or digital assets with other users, they connect to and then push their content to the shared server. This allows other users to fetch that content from the shared server and work with it on their own personal server. Users might need to merge content before pushing if their changes conflict with changes made by others.

In addition to supporting these two models, Helix Server also allows for a hybrid architecture in which some users connect directly to the shared server while others connect to personal servers that are connected to the shared server.

For more information about distributed development and file management, see *Using Helix Server for Distributed Versioning*.

Performance, scaling, and high availability

Version control systems are key to managing large projects: with Helix Server, “large” can be large indeed. With enterprise-level features that you can use to fine tune and improve performance, Helix Server lets you scale your system to accommodate a global workforce, and to automate failover for a highly available system. For example, Helix Server can accommodate the needs of a gaming development company whose files might take up hundreds of terabytes or even petabytes of data; or it can support the work of a software company, whose activity level includes massive automated testing as well as focused, analytic bug fixing and tracking work.

To support these tasks, Helix Server uses the following additional server types:

- *Proxies* are used where bandwidth to remote sites is limited; they mediate between remote clients and the versioning service. By caching frequently used files, the proxy reduces demand on the server and keeps network traffic to a minimum.
- *Brokers* mediate between clients and servers to implement policies that solve routing or security problems.
- *Replicas* duplicate server data. They can be used to provide a warm standby server or to reduce load on a primary server.

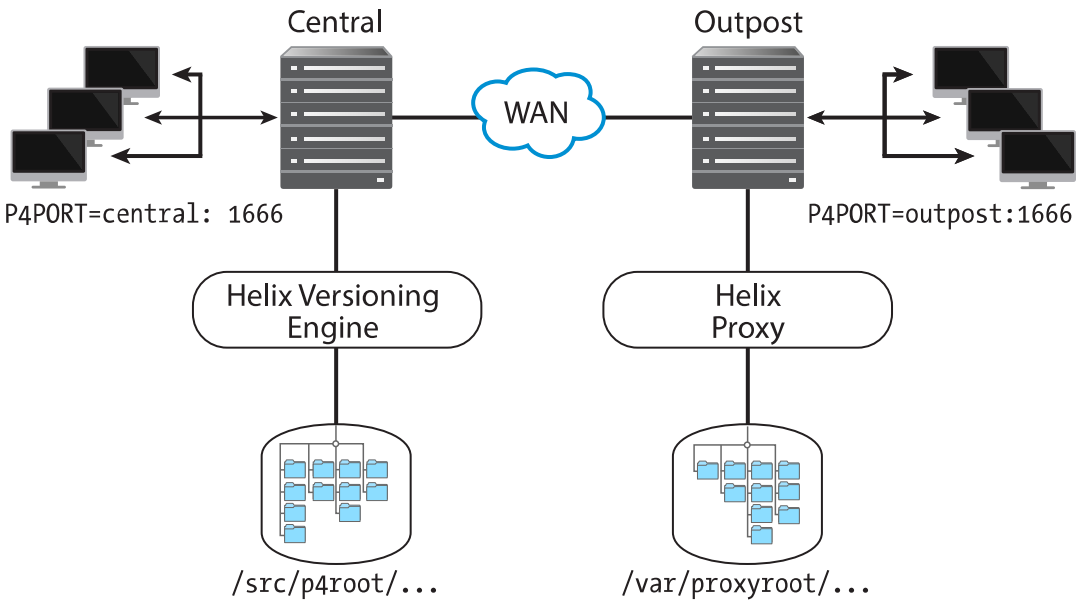
The following sections explain how these servers are used singly or how they are combined to provide enterprise-level performance. For complete information about using proxies, brokers, and replicas, see *Helix Versioning Engine Administrator Guide: Multi-Site Deployment*.

See also "Git at scale" on page 22.

Using proxies to improve performance

To improve performance for users accessing a shared Helix Server repository across a WAN, you can configure a proxy on the side of the network close to the users and configure the users to access the service through the proxy; then configure the proxy to access the master Helix Server service.

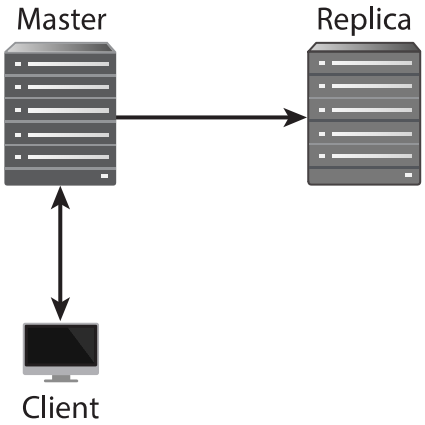
The following diagram illustrates a typical proxy configuration:



In this configuration, file revisions requested by users at a remote development site are fetched first from a shared 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; this architecture reduces both network traffic across the WAN and CPU load on the shared server.

Using a replica for disaster recovery

Replication is the duplication of server data from one Helix Server to another. The replicated server is called the master server; its replica is called a replica server. You can use replication to provide a warm standby server, or to reduce load and downtime on a primary server when performing builds. The following figure shows how you set up a replica to provide a warm standby to aid in disaster recovery.



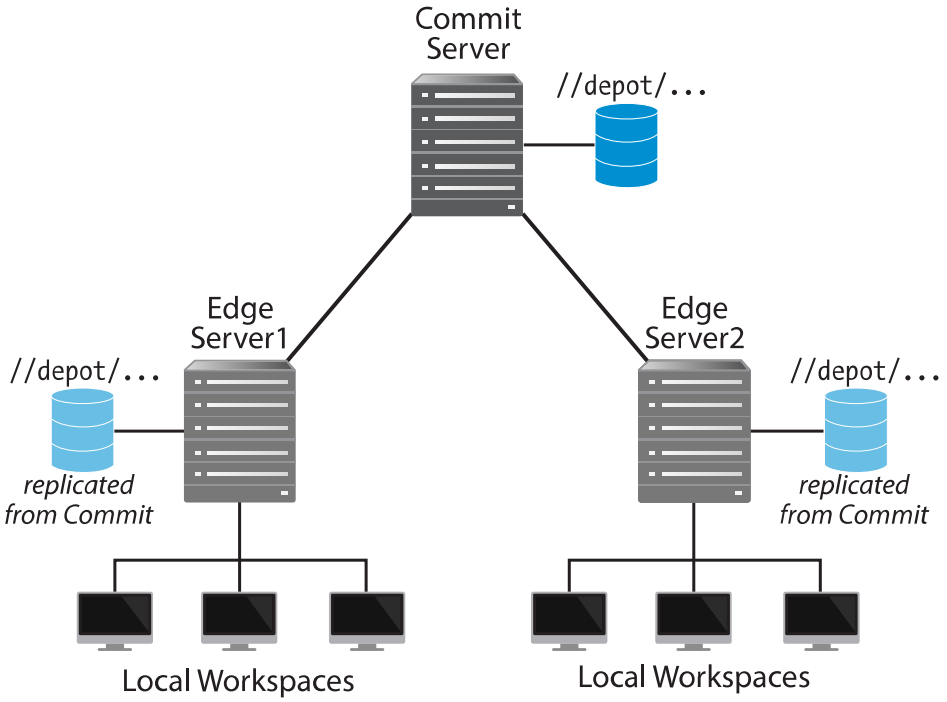
When you create the replica, you specify which server it should get its data from. The replica then periodically updates itself by copying files and metadata from the master. If the master fails, all you need do is reconfigure the replica to be the new master and then reconnect clients to communicate with it.

Edge servers and workspace servers, described in the following sections, are special examples of replica servers.

Commit-edge architecture

This architecture supports optimal performance for geographically distributed work groups. At a minimum it is made up of the following kinds of servers:

- A *commit* server that stores archives and metadata.
- 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. This server can handle read-only operations and operations that only write to the local data. You can connect multiple edge servers to a commit server as shown in the next figure.



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.

For more information about these options, see *Helix Versioning Engine Administrator Guide: Multi-Site Deployment*.

Securing the system

Independent of the Helix Server architecture you use, secure communication is guaranteed both with respect to communication between clients and servers as well as communication between servers.

- User authentications can be done using passwords or tickets, and the strength of the password can be defined by an administrator. Users can be authenticated against an Active Directory or LDAP server, or against an internal Helix Server user database.
- Communication between clients and servers can be secured using the SSL protocol, which you specify when connecting to the server.
- Communication between servers in a distributed environment can be secured using a trust file and by setting permissions for the service users that own the different servers in the environment.

In addition to user authentication, digital assets are further secured by a protection scheme to prevent unauthorized access. Protections determine which Helix Server commands can be run, on which files, by whom, and from which host. This scheme provides the finest grained control possible.

For more information on security, see the *Helix Versioning Engine Administrator Guide: Fundamentals* and the *Helix Versioning Engine Administrator Guide: Multi-Site Deployment*.

Clients, IDE's, builds

Perforce products also include tools and software packages that allow you to work with and analyze your data as well as seamlessly integrate Helix Server into interactive development environments and build solutions.

Helix client applications include:

- A command line interface available for all platforms
- A GUI interface for Mac OS X, UNIX, Linux, and Windows
- Integrations, or plug-ins, that work with commercial IDEs and productivity software

Interactive development environment integrations include:

- The Helix Plugin for Visual Studio, short P4VS, embeds the power of Helix Serverfeatures into the Microsoft Visual Studio IDE.
- The P4Eclipse plugin integrates the strengths of Helix Server with Eclipse's powerful IDE.
- P4Connect, the Helix Plugin for Unity, enables you to perform Helix Server operations directly from within Unity.

  You also have access to many community-built integration plugins, which are summarized on the Perforce website, on the Helix Plugins & Integrations page.

Build and reporting integrations include:

- The Jenkins integration allows Jenkins users to use Helix repositories directly. The plugin makes it simple to pull code, automate reviews, apply labels, and so on. Helix4Git can be part of a environment that uses Jenkins.
- The P42DB integration replicates Helix Server metadata to open source and commercial SQL databases.

Use cases

The Ace Rocket company develops software to manage the production and distribution of their space rockets and rocket launchers. They have a variety of technical needs, all of which can be met by members of the Helix Server product family. The following sections describe these needs and the Helix Server products that address them.

Software development

Ace Rocket uses a variety of programming languages to develop the components and products that they build. Because they use open source projects, they have numerous developers that use Git. For their internal development teams, they prefer the flexibility of Helix Server. Using Helix4Git, Ace Rocket can track their Git commits in the Helix Versioning Engine. This makes the Helix Versioning Engine a single source of truth for all of their assets and provides an unbroken history.

Ace's software developers use both Swarm to collaborate and communicate about the code they write. Ace Rocket's Windows developers take advantage of the native Microsoft Visual Studio integration (P4VS), which makes it easy for them to kick off code reviews from within their IDE. Ace's Java developers get the full power of Helix Server inside of their IDE of choice: Eclipse.

Using the Jenkins plugin, the Ace Rocket testing team continually tests code as it is checked into a single place by both Helix Server and Git developers.

Digital asset management

The Ace Rocket Company tracks design versions of their products using Helix Server. They track versions of their CAD/CAM files as well as large test datasets. Ace takes advantage of Helix Server's file locking to ensure that multiple people do not attempt to change a model at the same time.

Using configurable storage, Ace Rocket chooses to limit its storage consumption to only 20 versions of each file. Using Helix Server's multi-site deployment architecture, Ace seamlessly replicates these large files across the globe to improve access times for their engineers.

Hybrid product development

With both source code and CAD/CAM files tracked in Helix Server, Ace Rocket can easily track dependencies between their code and their models. Developers take advantage of Helix Server's fluid distributed workflows, while Ace's model engineers use locking to obtain the tight control they need to collaborate with each other.

Ace Rocket's IT team takes advantage of the numerous Helix Server APIs to build custom tools that meet their unique needs and to integrate Helix Server into all of their systems. In addition, they enjoy the benefits of Helix Server's native Active Directory support to make it easy to manage all of their users.

Another type of hybrid product development is combining Helix4Git with classic Helix Server. See Helix4Git Administration.

Ease of use for Helix Server administration

Ace Rocket employs an outside agency for contractors. They want to give these contractors access to only a certain subset of the depot, while they want to give in-house developers full access. Because Helix Server permissions are highly configurable, it's easy for an administrator to use finely grained access control to manage access for different types of users.

Growing into the future

As Ace Rocket grows and develops offices in Shanghai, Buenos Aires, and Johannesburg, they deploy edge servers and replicas to support thousands of developers in these remote offices. Both developers and digital asset managers in remote offices benefit from having local access to their software or assets.

Helix Server is keeping one unified view, worldwide, of digital assets content and history so the distributed teams can collectively work as one.

Git at scale

Ace Rocket benefits from having scalable Git environments in **one** place. Helix4Git, with its Git Connector component, enables Ace Rocket to have rapid mirroring of Git repos into Helix Server, and supports real-time visibility into Ace Rocket's Continuous Integration (CI) activities with an automated build tool, such as Jenkins.

Support for Global Teams

- Ace Rocket enjoys good performance and usability across distributed geographic locations.
- Ace Rocket's enterprise teams have fast and stable development environments, wherever they are in the world.

Reduced Overhead and Tooling

- No danger of repo sprawl.
- Scalability for large digital assets, large numbers of files, and large total repo sizes are supported with streamlined repository management.

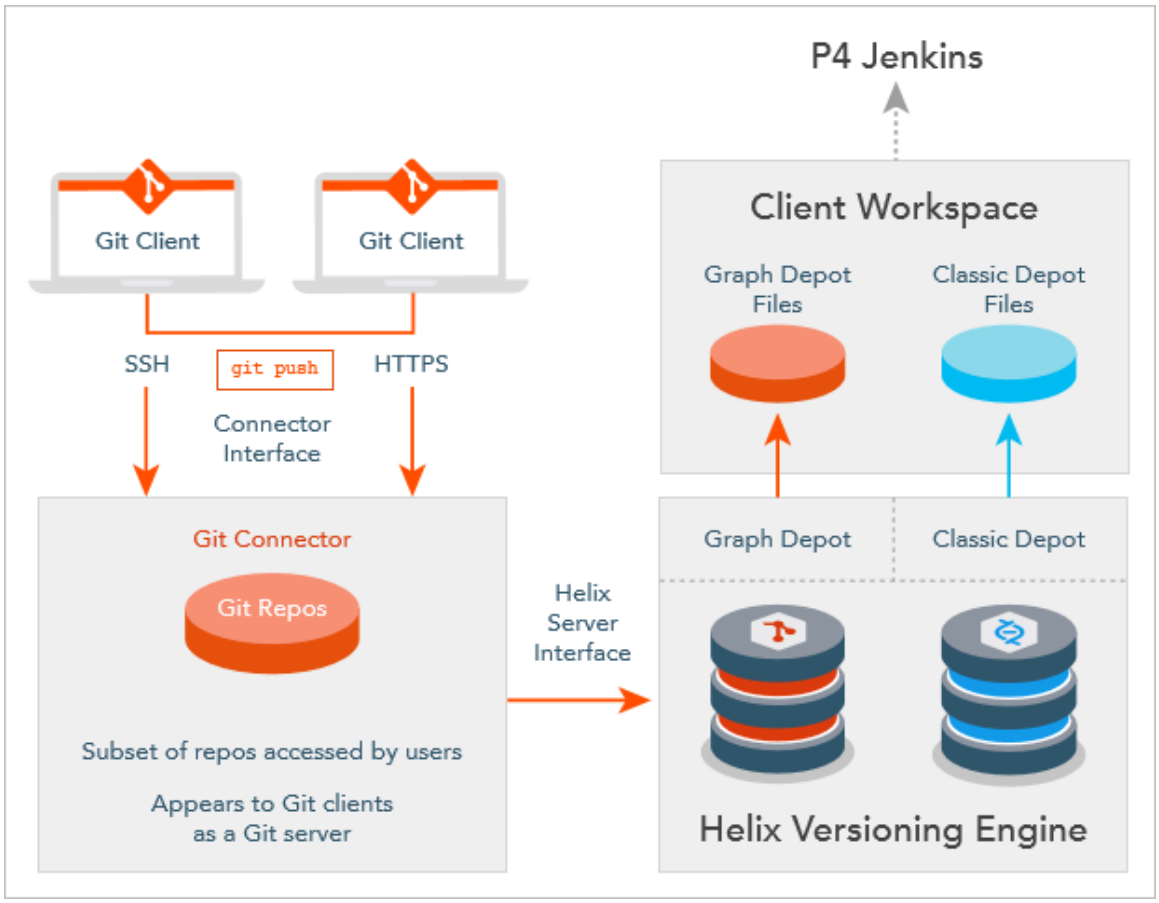
Centrally Manage All Digital Assets

- The graph depot accepts repos from multiple sources.
- Users can centrally manage their assets with multi-repo support and traceability.

Git Continuous Integration and Delivery

- Automatic mirroring into Helix Server from third-party Git servers, such as GitHub, GitLab, and Gerrit, help Ace Rocket achieve rapid release iterations and development benchmarks.
- Helix4Git boosts performance with simplified builds of multi-repo, multi-asset projects on top of a federated architecture.

For more information, see https://www.perforce.com/git-scale.



To learn more about Helix Server

To ...	See ...
Download Helix products	http://www.perforce.com/downloads
View tutorials explaining how Helix products work	https://www.perforce.com/support/video-tutorials
Get online help from within Helix Server applications	■ Use the help menu from within graphical Perforce applications ■ Type `p4 help` from the command line for help with the Command-Line Client
Access the Perforce Knowledge Base, which has Support articles	http://answers.perforce.com
Access the Perforce Forums where users ask and answer questions	http://forums.perforce.com/
Request Support by phone or email	https://www.perforce.com/support

Documentation for users, developers, and administrators is at https://www.perforce.com/support/self-service-resources/documentation.

Glossary

A

access level

A permission assigned to a user to control which commands the user can execute. See also the 'protections' entry in this glossary and the 'p4 protect' command in the P4 Command Reference.

admin access

An access level that gives the user permission to privileged commands, usually super privileges.

archive

1. For replication, versioned files (as opposed to database metadata). 2. For the 'p4 archive' command, a special depot in which to copy the server data (ersioned files and metadata).

atomic change transaction

Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.

B

base

The file revision, in conjunction with the source revision, used to help determine what integration changes should be applied to the target revision.

binary file type

A Helix Server file type assigned to a non-text file. By default, the contents of each revision are stored in full, and file revision is stored in compressed format.

branch

(noun) A set of related files that exist at a specific location in the Perforce depot as a result of being copied to that location, as opposed to being added to that location. A group of related files is often referred to as a codeline. (verb) To create a codeline by copying another codeline with the 'p4 integrate', 'p4 copy', or 'p4 populate' command.

branch form

The form that appears when you use the 'p4 branch' command to create or modify a branch specification.

branch mapping

Specifies how a branch is to be created or integrated by defining the location, the files, and the exclusions of the original codeline and the target codeline. The branch mapping is used by the integration process to create and update branches.

branch view

A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. This is the same as branch mapping.

broker

Helix Broker, a server process that intercepts commands to the Helix Server and is able to run scripts on the commands before sending them to the Helix Server.

C

change review

The process of sending email to users who have registered their interest in changelists that include specified files in the depot.

changelist

A list of files, their version numbers, the changes made to the files, and a description of the changes made. A changelist is the basic unit of versioned work in Helix Server. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot. See also atomic change transaction.

changelist form

The form that appears when you modify a changelist using the 'p4 change' command.

changelist number

The unique numeric identifier of a changelist. By default, changelists are sequential.

check in

To submit a file to the Helix Server depot.

check out

To designate one or more files for edit.

checkpoint

A backup copy of the underlying metadata at a particular moment in time. A checkpoint can recreate db.user, db.protect, and other db.* files. See also metadata.

classic depot

A repository of Helix Server files that is not streams-based. The default depot name is depot. See also default depot and stream depot.

client form

The form you use to define a client workspace, such as with the 'p4 client' or 'p4 workspace' commands.

client name

A name that uniquely identifies the current client workspace. Client workspaces, labels, and branch specifications cannot share the same name.

client root

The topmost (root) directory of a client workspace. If two or more client workspaces are located on one machine, they should not share a client root directory.

client side

The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.

client workspace

Directories on your machine where you work on file revisions that are managed by Helix Server. By default, this name is set to the name of the machine on which your client workspace is located, but it can be overridden. Client workspaces, labels, and branch specifications cannot share the same name.

code review

A process in Helix Swarm by which other developers can see your code, provide feedback, and approve or reject your changes.

codeline

A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

comment

Feedback provided in Helix Swarm on a changelist or a file within a change.

commit server

A server that is part of an edge/commit system that processes submitted files (checkins), global workspaces, and promoted shelves.

conflict

1. A situation where two users open the same file for edit. One user submits the file, after which the other user cannot submit unless the file is resolved. 2. A resolve where the same line is changed when merging one file into another. This type of conflict occurs when the comparison of two files to a base yields different results, indicating that the files have been changed in different ways. In this case, the merge cannot be done automatically and must be resolved manually. See file conflict.

copy up

A Helix Server best practice to copy (and not merge) changes from less stable lines to more stable lines. See also merge.

counter

A numeric variable used to track variables such as changelists, checkpoints, and reviews.

D

default changelist

The changelist used by a file add, edit, or delete, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.

deleted file

In Helix Server, a file with its head revision marked as deleted. Older revisions of the file are still available. in Helix Server, a deleted file is simply another revision of the file.

delta

The differences between two files.

depot

A file repository hosted on the server. A depot is the top-level unit of storage for versioned files (depot files or source files) within a Helix Versioning Engine. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

The left side of any client view mapping, specifying the location of files in a depot.

depot syntax

Helix Server syntax for specifying the location of files in the depot. Depot syntax begins with: //depot/

diff

(noun) A set of lines that do not match when two files are compared. A conflict is a pair of unequal diffs between each of two files and a base. (verb) To compare the contents of files or file revisions. See also conflict.

donor file

The file from which changes are taken when propagating changes from one file to another.

E

edge server

A replica server that is part of an edge/commit system that is able to process most read/write commands, including 'p4 integrate', and also deliver versioned files (depot files).

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

F

file conflict

In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also, an attempt to submit a file that is not an edit of the head revision of the file in the depot, which typically occurs when another user opens the file for edit after you have opened the file for edit.

file pattern

Helix Server command line syntax that enables you to specify files using wildcards.

file repository

The master copy of all files, which is shared by all users. In Helix Server, this is called the depot.

file revision

A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, preceded by a pound sign (#), for example testfile#3.

file tree

All the subdirectories and files under a given root directory.

file type

An attribute that determines how Helix Server stores and diffs a particular file. Examples of file types are text and binary.

fix

A job that has been closed in a changelist.

form

A screen displayed by certain Helix Server commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

A replica server that can process read-only commands and deliver versioned files (depot files). One or more replicat servers can significantly improve performance by offloading some of the master server load. In many cases, a forwarding replica can become a disaster recovery server.

G

Git Fusion

A Perforce product that integrates Git with Helix, offering enterprise-ready Git repository management, and workflows that allow Git and Helix Server users to collaborate on the same projects using their preferred tools.

graph depot

A depot of type graph that is used to store Git repos in the Helix Server. See also Helix4Git.

group

A feature in Helix Server that makes it easier to manage permissions for multiple users.

H

have list

The list of file revisions currently in the client workspace.

head revision

The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.

Helix Server

The Helix Server depot and metadata; also, the program that manages the depot and metadata, also called Helix Versioning Engine.

Helix TeamHub

A Perforce management platform for code and artifact repository. TeamHub offers built-in support for Git, SVN, Mercurial, Maven, and more.

Helix4Git

Perforce solution for teams using Git. Helix4Git offers both speed and scalability and supports hybrid environments consisting of Git repositories and 'classic' Helix Server depots.

I

integrate

To compare two sets of files (for example, two codeline branches) and determine which changes in one set apply to the other, determine if the changes have already been propagated, and propagate any outstanding changes from one set to another.

J

job

A user-defined unit of work tracked by Helix Server. The job template determines what information is tracked. The template can be modified by the Helix Server system administrator. A job describes work to be done, such as a bug fix. Associating a job with a changelist records which changes fixed the bug.

job specification

A form describing the fields and possible values for each job stored in the Helix Server machine.

job view

A syntax used for searching Helix Server jobs.

journal

A file containing a record of every change made to the Helix Server's metadata since the time of the last checkpoint. This file grows as each Helix Server transaction is logged. The file should be automatically truncated and renamed intoa numbered journal when a checkpoint is taken.

journal rotation

The process of renaming the current journal to a numbered journal file.

journaling

The process of recording changes made to the Helix Server's metadata.

L

label

A named list of user-specified file revisions.

label view

The view that specifies which filenames in the depot can be stored in a particular label.

lazy copy

A method used by Helix Server to make internal copies of files without duplicating file content in the depot. A lazy copy points to the original versioned file (depot file). Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

license file

A file that ensures that the number of Helix Server users on your site does not exceed the number for which you have paid.

list access

A protection level that enables you to run reporting commands but prevents access to the contents of files.

local depot

Any depot located on the currently specified Helix Server.

local syntax

The syntax for specifying a filename that is specific to an operating system.

lock

1. A file lock that prevents other clients from submitting the locked file. Files are unlocked with the 'p4 unlock' command or by submitting the changelist that contains the locked file. 2. A database lock that prevents another process from modifying the database db.* file.

log

Error output from the Helix Server. To specify a log file, set the P4LOG environment variable or use the p4d -L flag when starting the service.

M

mapping

A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. See also workspace view, branch view, and label view.

MDS checksum

The method used by Helix Server to verify the integrity of versioned files (depot files).

merge

1. To create new files from existing files, preserving their ancestry (branching). 2. To propagate changes from one set of files to another. 3. The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.

merge file

A file generated by the Helix Server from two conflicting file revisions.

metadata

The data stored by the Helix Server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata includes all the data stored in the Perforce service except for the actual contents of the files.

modification time or modtime

The time a file was last changed.

N

nonexistent revision

A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the #none revision specifier are examples of nonexistent file revisions.

numbered changelist

A pending changelist to which Helix Server has assigned a number.

O

opened file

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

owner

The Helix Server user who created a particular client, branch, or label.

P

p4

1. The Helix Versioning Engine command line program. 2. The command you issue to execute commands from the operating system command line.

p4d

The program that runs the Helix Server; p4d manages depot files and metadata.

pending changelist

A changelist that has not been submitted.

project

In Helix Swarm, a group of Helix Server users who are working together on a specific codebase, defined by one or more branches of code, along with options for a job filter, automated test integration, and automated deployment.

protections

The permissions stored in the Helix Server's protections table.

proxy server

A Helix Server that stores versioned files. A proxy server does not perform any commands. It serves versioned files to Helix Server clients.

R

RCS format

Revision Control System format. Used for storing revisions of text files in versioned files (depot files). RCS format uses reverse delta encoding for file storage. Helix Server uses RCS format to store text files. See also reverse delta storage.

read access

A protection level that enables you to read the contents of files managed by Helix Server but not make any changes.

remote depot

A depot located on another Helix Server accessed by the current Helix Server.

replica

A Helix Server that contains a full or partial copy of metadata from a master Helix Server. Replica servers are typically updated every second to stay synchronized with the master server.

repo

A graph depot contains one or more repos, and each repo contains files from Git users.

reresolve

The process of resolving a file after the file is resolved and before it is submitted.

resolve

The process you use to manage the differences between two revisions of a file. You can choose to resolve conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes.

reverse delta storage

The method that Helix Server uses to store revisions of text files. Helix Server stores the changes between each revision and its previous revision, plus the full text of the head revision.

revert

To discard the changes you have made to a file in the client workspace before a submit.

review access

A special protections level that includes read and list accesses and grants permission to run the p4 review command.

revision number

A number indicating which revision of the file is being referred to, typically designated with a pound sign (#).

revision range

A range of revision numbers for a specified file, specified as the low and high end of the range. For example, myfile#5,7 specifies revisions 5 through 7 of myfile.

revision specification

A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, a revision range, change numbers, label names, date/time specifications, or client names.

S

server data

The combination of server metadata (the Helix Server database) and the depot files (your organization's versioned source code and binary assets).

server root

The topmost directory in which p4d stores its metadata (db.* files) and all versioned files (depot files or source files). To specify the server root, set the P4ROOT environment variable or use the p4d -r flag.

service

In the Helix Versioning Engine, the shared versioning service that responds to requests from Helix Server client applications. The Helix Server (p4d) maintains depot files and metadata describing the files and also tracks the state of client workspaces.

shelve

The process of temporarily storing files in the Helix Server without checking in a changelist.

status

For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses. For the 'p4 status' command, by default the files opened and the files that need to be reconciled.

stream

A branch with additional intelligence that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients.

submit

To send a pending changelist into the Helix Server depot for processing.

super access

An access level that gives the user permission to run every Helix Server command, including commands that set protections, install triggers, or shut down the service for maintenance.

symlink file type

A Helix Server file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.

sync

To copy a file revision (or set of file revisions) from the Helix Server depot to a client workspace.

T

target file

The file that receives the changes from the donor file when you integrate changes between two codelines.

text file type

Helix Server file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

The revision in the depot with which the client file (your file) is merged when you resolve a file conflict. When you are working with branched files, theirs is the donor file.

three-way merge

The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

trigger

A script automatically invoked by Helix Server when various conditions are met. (See "Helix Versioning Engine Administrator Guide: Fundamentals" on "Using triggers to customize behavior")

two-way merge

The process of combining two file revisions. In a two-way merge, you can see differences between the files.

typemap

A table in Helix Server in which you assign file types to files.

U

user

The identifier that Helix Server uses to determine who is performing an operation.

V

versioned file

Source files stored in the Helix Server depot, including one or more revisions. Also known as a depot file or source file. Versioned files typically use the naming convention 'filenamev' or '1.changelist.gz'.

view

A description of the relationship between two sets of files. See workspace view, label view, branch view.

W

wildcard

A special character used to match other characters in strings. The following wildcards are available in Helix Server: * matches anything except a slash; ... matches anything including slashes; %%0 through %%9 is used for parameter substitution in views.

workspace

See client workspace.

workspace view

A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.

write access

A protection level that enables you to run commands that alter the contents of files in the depot. Write access includes read and list accesses.

Y

yours

The edited version of a file in your client workspace when you resolve a file. Also, the target file when you integrate a branched file.

License Statements

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/).