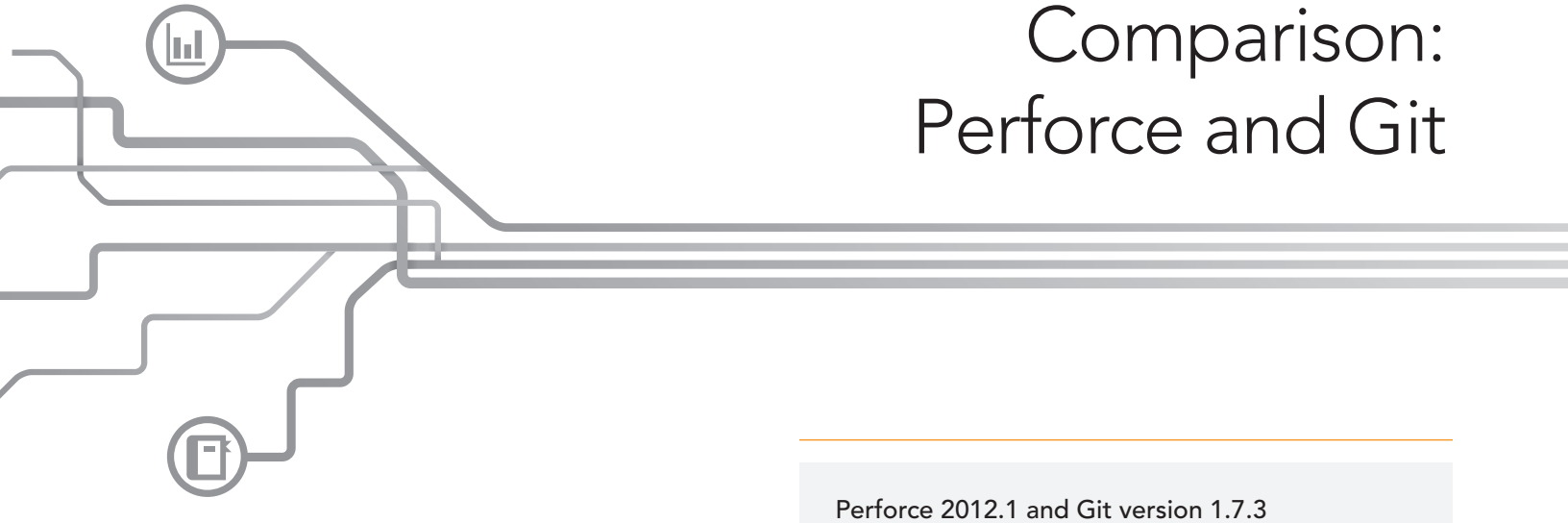
PERFORCE
Version everything.



Comparison: Perforce and Git

Perforce 2012.1 and Git version 1.7.3

This document compares Perforce (version 2012.1) and Git version 1.7.3. Read this comparison to:

- Understand the major feature differences
- See how Perforce and Git compare on qualitative aspects such as usability and administration
- Get a general comparison of the effects of scaling on both systems

Table of Contents

Executive Summary ____ 5

Overview ____ 5

A Word on Architecture ____ 10

Working Effectively ____ 6

Branching and Merging

Branch Creation and Usage ____ 6

Sparse Branching ____ 6

Merging Changes ____ 6

Cherry Picking Revisions for a Merge ____ 8

Renaming and Refactoring Files ____ 8

Task-based Work ____ 9

Switching Tasks and Storing Work-in-Progress ____ 9

Link to Task Management ____ 9

Workspace Model ____ 9

Ease of Use ____ 9

Graphical Applications and Integrations ____ 9

Identify Changes ____ 9

Collaboration ____ 9

Code Sharing and Dependency Management ____ 9

Perforce Views and Streams ____ 9

Git Sub-modules and Sub-trees ____ 9

Branching and Release Management ____ 9

Project Creation ____ 9

Promoting Work ____ 9

Understanding The Branch Model ____ 9

A Release Management Scenario ____ 9

The Scenario in Perforce ____ 9

The Scenario in Git ____ 9

Digital Assets That Cannot Be Merged ____ 9

Visual Tools for Collaboration ____ 9

Application Lifecycle Management (ALM) and Community Development ____ 9

Scalability 10
Large File Management 10
Cross-Platform Compatibility 10
Remote Development 10
Extensions: APIs and Scripting 10

Administration and Management
Setup and Deployment 10
Security, Authentication, and Access Control 10
Auditing 6
Backup and Maintenance 10
The Basics 6
Repository Verification and Maintenance 6
Recovering Lost Workspaces 6
Policy Management 10
Triggers 6
Reporting and Data Mining 6
Support and Services 6

Conclusion

Learn More
Evaluating Perforce 10
Scheduling a Demo of Perforce 10
Migrating to Perforce 10

EXECUTIVE SUMMARY

The traditional strengths of Perforce as a shared software version management system satisfy the requirements of the enterprise environment: release management, scalability, global collaboration, and security are all vital to the enterprise, or indeed most modern development teams.

More recently, distributed version control systems (DVCSs)—most notably Git—have gained adoption in other usage models, including traditional enterprise environments. DVCSs started to appear in 2001. These tools were originally designed for the open source usage model: widely dispersed collaborators working mostly independently, but sharing work on a limited basis.

Although the architecture of a DVCS is interesting from a technical perspective, the workflow aspects are more important, as they have highlighted unsolved problems for many individual users.

Intensive research has revealed that the most compelling reasons for adoption of a DVCS are:

- Private local branching, or the ability to work freely outside the constraints of the enterprise version control model. This autonomy spurs creativity and results in better work.
- A quick and simple workflow for common tasks.
- Connection independent versioning, or the ability to work effectively with a slow or non-existent connection to the rest of the enterprise.

Perforce has studied how the problems highlighted by DVCS can be solved effectively. Rather than starting from scratch or attempting to add enterprise features to DVCS, combining the power of shared software version management with the flexibility of DVCS offers the best solution to the enterprise and to individual users. And that is the ultimate goal: providing a tool and a model that facilitate effective work.

In this paper we attempt to analyze the solutions offered by Perforce and Git, a popular DVCS. Focusing on qualitative aspects such as usability and administration, we compare Perforce version 2012.1 with Git version 1.7.3.

OVERVIEW

Attribute	Git	Perforce
Working Effectively **Perforce (via P4Sandbox) and Git offer powerful workflow and distributed working solutions.**		
Branching and Merging	Quick and effective private branching. Rebasing provides extra flexibility for private work.	P4Sandbox supports easy private branch creation and use. Streams provide guidance for branching operations. Powerful merge GUI. More granular and flexible branching operations.
Task-based Work	Easy task branching and branch switching. Work-in-progress can be stashed but not easily shared. Some task management integration.	Easy task branching, branch switching, and automatic saving of work-in-progress. Shelves offer simple task hand-off and review. Integrates easily with task management.
Workspace Model	A workspace is a simple view of an entire repository.	Supports flexible workspace views.
Ease of Use	Several GUI clients available for different platforms, with limited feature sets, targeted at technical users. Globally unique (but non-intuitive) revision identifiers. Simple tags (labels) can be used at the repository level.	Powerful visual tools available for technical and non-technical users. Easy organizational and identification techniques, including flexible labels.

Attribute	Git	Perforce
Collaboration **Perforce offers more powerful and flexible tools for collaboration, which allow release managers and product architects to guide the work of the team.**		
Code Sharing	Sub-modules allow code sharing, but require extra planning and disciplined usage.	Several methods, notably streams, available to easily share code between projects and manage module dependencies. Relationships between modules can be guided by the project architect.
Branching and Release Management	Creating a new project requires creating a new repository. More work required to model the overall release management process.	Several projects can be hosted in a single server, simplifying release management. Streams provide a powerful framework for guiding collaboration.
Working on Files That Cannot be Merged	No support for file locking or concurrent edit notifications.	Supports file locking and concurrent edit notifications in a single branch.
Visual Tools to Aid Collaboration	Visual tools display information about a single repository.	Visual tools provide rich contextual information about the project(s) of interest.
ALM and Community Development	Community sites provide ALM-like features and management tools.	Superior collaboration tools for teams. Integrates into any ALM suite and provides own management tools. Free for open source use.
Scalability **Perforce scales to manage thousands of users and terabytes of data, while still facilitating collaboration. Scaling Git requires extra systems and tools to integrate and maintain.**		
Large Files	Very poor handling of large files without additional systems.	Efficient storage and use of large files.
Cross-platform Compatibility	Standard or community supported distributions available for all major platforms. Uneven Windows support.	Supports all major platforms. Services and applications interoperate across platforms.
Distributed Development	Excellent support for distributed work.	P4Sandbox offers excellent support for distributed private work. Perforce has several tools to sustain distributed development for the shared repository.
Extensions	Stable C++ API. Many extensions built around command line interface and API.	Supported APIs for many platforms and languages, including web services.

Attribute	Git	Perforce
Administration and Management **Perforce provides reliable management tools and support. Basic Git administration is straightforward, but additional work or tools required in a team setting. Git support provided by community.**		
Setup and Deployment	Administration involves coordination with IT and individual repository maintainers. Release management strategy affects deployment.	Straightforward architecture with minimal infrastructure requirements. A single administrator can maintain Perforce for hundreds of users.
Security and Access Control	Authentication done via HTTP server or operating system. Access control is done at the repository level, which impacts the development model.	Centrally managed, granular authentication and access control mechanisms. Scales well for sites of any size. Auditing built in.
Backup and Maintenance	Backing up a single repository is simple. Each repository in the deployment must be backed up and maintained, although cloned repositories offer some backup capability by default. Backups must include hooks, security configuration, and other metadata.	Well-established, centrally administered backup, recovery, and maintenance procedures.
Policy Management	Supports hooks but relies more on upstream committers to guide or enforce policy across an entire team.	Triggers and broker provide easy ways to provide structure and guidance to a team.
Reporting and Data Mining	Simple reporting for a single project (repository).	Rich reporting on all aspects of all projects. Sophisticated data mining available via report engines.
Support and Services	Supported via community.	World-class technical support and related services.

A Word on Architecture

Although the architecture of a software version management system is not of primary concern to most users, the architecture of a DVCS has important usage implications.

A software version management system such as Perforce uses a single canonical representation of the repository's data. All users connect to a shared versioning service which updates this canonical data set. The deployment architecture may contain tools such as replication services to help with performance or support, but fundamentally there is still a single copy of important data.

Perforce uniquely offers P4Sandbox, a tool that supports independent work with tight integration to the shared repository.

A DVCS like Git uses a single, independent repository for each workspace. Often a DVCS is designed such that a single repository contains a single project's data, and most actions operate on the entire repository. In a typical workflow, one person maintains a repository that accepts changes from contributors. Each contributor will start by cloning a complete copy of that repository.

A single contributor will make changes privately and push them upstream when ready (or request that the shared repository maintainer pull their changes).

Pro Git (Apress, 2009) has an excellent starting discussion on the usual Git workflows. There is no definitive deployment diagram for Git, but a sample deployment or workflow architecture is shown in Figure 6.

WORKING EFFECTIVELY

Software version management provides the tools and framework for users to understand and evolve their digital assets.

In this section we consider how Perforce and Git help users do their work effectively at the private or local level.

Branching and Merging

Branching and merging is the process of creating new copies of digital assets, working on them concurrently, and at some point sharing changes between those copies. Branching and merging can occur privately to facilitate working on specific tasks or just to experiment, or publicly, to allow teams to work in parallel and collaborate.

The ability to freely create and use private branches has proven to be a powerful productivity boost: users are able to experiment, be creative, and use branches to manage work on several tasks. Work in progress is committed frequently, and different approaches to a problem are tried. Work does not need to be shared until it reaches a point of maturity.

This autonomy has led to more productive work. In this section we discuss the ease of creating and using branches at the private (local or individual) level.

Branch Creation and Usage

The mechanics of creating and using a new branch in Git are very simple. Branches exist in the Git metadata but are not directly reflected in the local directory structure. Git's isolation of

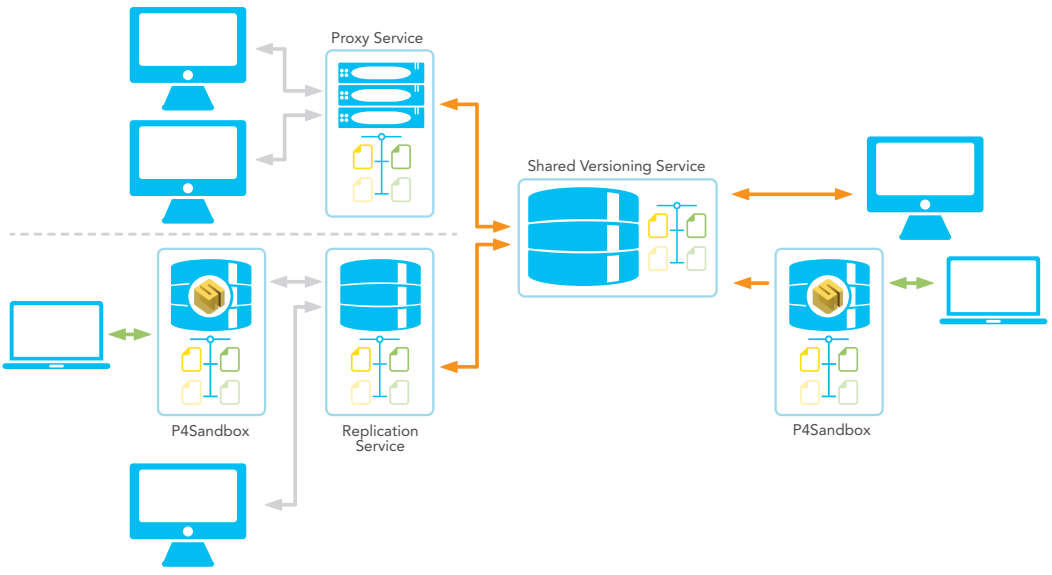


Figure 1: Perforce deployment architecture

branches from the directory tree makes it easy to delete unwanted branches. a task more commonly done at the private rather than at the team or enterprise level.

The relationship between branches in a Git repository is purely a matter of convention; all branches have equal weight in Git.

Most Git commands operate on the entire repository, so there is little ability to branch and work on only a subset of files, or create new views of the directory tree. Git does have the ability to checkout only a subset of a repository, but most commands are not granular in scope.

In Perforce, the mechanics of creating and using a new branch are straightforward. With the Perforce Streams framework introduced in Perforce 2011.1, the process is simplified, and can be done quickly and intuitively in the associated visual tools.

Branches exist in the Perforce metadata and are reflected in the directory structure. A branch in Perforce is, from one perspective, just another directory in the workspace. If we choose, we can see as many branches of as many projects in our workspace as we like. (Of course, if we prefer to have a workspace only contain a single branch, we can create new workspaces for new branches.) A good naming convention for branches will immediately expose some structure in the directory layout. When working in the Perforce Streams framework, the relationship between branches is also codified. (see Understanding the Branch Model).

Perforce, by virtue of branch and stream views, more easily supports creating complex branches. We can use branch mappings, for example, to move all documentation files into a different location when creating a release branch. Or, we can use the virtual module technique to work with only a subset of a branch for bug fixing (see Perforce Views and Streams).

In a shared repository, there are often policy restrictions governing the creation and usage of new branches. Branch location, ownership, and usage must be determined in the context of the overall release management framework. Additionally, the proliferation of branches in a shared repository may have performance impacts. Because of these reasons, users often are not free to create new branches at will. The new P4Sandbox product provides the extra layer of isolation that allows Perforce users to create and use private branches easily without impacting the shared service.

Sparse Branching

Sparse branches are constructed by only branching the files actually modified, and otherwise using files from

the parent branch. Sparse branching keeps branches small and branching operations fast.

Git supports the effective equivalent of sparse branching by default. Branch creation is simply a reference in the Git metadata; new file revisions are only created upon commit of actual changes. Git's lightweight branches are part of the reason that Git branching is considered fast and easy.

Perforce branches are fully populated by default on the shared service, which is perhaps a simpler model when collaboration between several users is intended. When working in P4Sandbox, a just-in-time branching process is used that is very similar to sparse branching. Branching in P4Sandbox as a result is also very fast and easy.

Merging Changes

Merging or propagating change is necessary both at the private (local) and public levels. Both Perforce and Git handle common merge tasks easily, although Perforce provides a powerful graphical merge tool, P4Merge, when manual review is necessary.

Perforce's Streams framework provides structure and guidance for merging changes (see Understanding the Branch Model). Perforce also provides more flexibility when choosing how to merge changes. Perforce's two-phase integrate and resolve process allows developers to run a merge, then choose the resolve action (merge strategy) that is most appropriate for each file.

Git, on the other hand, requires choosing a merge strategy in advance for the entire branch.

Cherry Picking Revisions for a Merge

Cherry picking revisions for a merge is not a best practice by any means, but is sometimes necessary. The most common scenario involves an urgent need to promote one piece of new development code to a release maintenance branch. Perforce gives you complete control over which revisions to include in a merge, using any of its revision specifiers using the `p4 integ` command.

Git provides two very different commands to use to perform a selective merge. The `git cherry-pick` command works on a single revision, and there is a detailed formula involving the `git rebase` command that works on ranges of revisions.

It is important to note the distinction between Perforce's integration command and Git's rebase operation. Perforce's integration command always understands and records merge history. Subsequent merges after cherry-picking will know, and respect, that a cherry-pick was done.

Git rebase does not impact merge history; it is more similar to replaying a set of edits after updating a branch to a new starting point. Git's rebase operation is best used in isolation. Rebasing after pushing changes upstream will lead to a future complex merge.

This behavior is due to one of Git's core use cases, selectively integrating patches into a master repository. In this situation, it is actually not desired to properly record merge history between the master repository and the contributing repository.

Renaming and Refactoring Files

Renaming a file often causes a problematic situation during a merge. Let's say that we have a file named `edit.c` on trunk, and we've renamed it to `write.c` on a development branch. When merging between the two branches in the future, there are several ways we might want the software version management system to handle the relationship between these files.

- Propagate the rename to trunk.
- Do not propagate the rename, but continue to propagate changes between `edit.c` on trunk and `write.c` on the development branch.
- Ignore any future changes to either file when merging between these branches.

Perforce gives us the flexibility to choose any of these options. The default behavior is to propagate the rename, but we can use the branch or stream view to choose another option. This flexibility is due to Perforce Inter-File Branching.

Git also supports renaming or moving a file. The `git merge` command detects this operation and handles it by propagating the rename. However, it does not provide a mechanism to ignore changes for this file. Propagating changes between the new and old names is possible, but only if the rename action is ignored by merging that change with the "ours" merge strategy. Subsequent merges will then propagate changes between the old and new names. Since a merge operates on an entire branch, renames must be handled carefully if the default Git handling is not desired; otherwise the "ours" merge strategy will ignore other changes as well.

Task-based Work

Task-focused development enhances productivity, and is particularly useful for Agile development methodologies that organize work around stories and tasks.

Switching Tasks and Storing Work-in-Progress

Both Perforce (via P4Sandbox) and Git provide excellent support for quickly updating a local workspace to focus on a new task. Creating new private branches to work on different tasks is very fast. Switching between these branches is easy, as both Perforce and Git offer in-place branching.

P4Sandbox also offers the ability to automatically save work-in-progress via Perforce's shelving feature. That makes switching tasks (branches) even easier.

Perforce's shelving feature is more powerful in many respects than Git's `stash` mechanism. Perforce shelves can be shared with other users (when not working in P4Sandbox), offering a simple code review and task hand-off capability. Git stashes offer a distinctive queue workflow that is appealing in ways, but sharing work in progress requires distribution via patches or published branches.

Link to Task Management

Task management (defect tracking or project management) tools provide the context to understand commits in a software version management system: why the change is being made and how it relates to requirements, test cases, and QA plans. Establishing the link between the software version management system and task management is important for improving transparency and collaboration.

Perforce, via its jobs system, provides a reliable and intuitive way to link the software version management system information to tasks. The Perforce Defect Tracking Gateway provides replication of information between Perforce and popular tools like HP Quality Center, JIRA, Redmine, and Bugzilla. Other integrations are offered for similar tools.

Git relies on community supported plugins to link to task management, and these plugins often rely on commit comment scanning, which is unreliable.

Workspace Model

Perforce allows very flexible views in application workspaces. The workspace view defines the files visible in the workspace and the location of those files on the local file system. Since each Git workspace is a clone of the entire repository, the user has less granular control over the composition of a workspace.

Users can easily rearrange files in their workspace to match the requirements of a build tool, for instance. Consider a Java development effort that started with this directory structure:

```
- src/java
- test/java
- doc/license
```

On one platform we need to compile this project using Maven, which dictates a different directory structure:

```
- src/main/java
- src/test/java
- license.txt
```

Rearranging the directory structure this way using a Perforce workspace or stream view is trivial, does not require a commit, and can be easily reproduced for other users. Changes made to the files under the new directory structure would transparently flow back to the files in the original location in the repository.

Achieving a similar goal with Git would be difficult without resorting to file system links, which are difficult to deploy automatically. Git's `read-tree` and `filter-branch` commands provide a directory mapping function, but are not considered typical end-user commands, and do not allow pushing changes transparently between the files in the original directory structure and the remapped copies.

Ease of Use

Modern software version management tools are quickly becoming repositories for more than purely technical data. The software version management system, by virtue of its file handling and security features, lends itself readily to managing a broader set of documents than just source code. The software version management system is commonly used to store documentation for products and processes, and may well be the only system available that can satisfy the audit requirements of defense or Sarbanes-Oxley environments. Even in distributed open source communities, an accessible software version management system will encourage contributions from documentation writers, graphic artists, translators, and other segments of the community.

Given that the users of software version management system data may well be outside of the technical role, it should be accessible to novice and non-technical users.

Graphical Applications and Integrations

Graphical applications (GUIs) are an important tool for most users. For less technically oriented users, a GUI may be the only application that will ever be truly useful and accessible. For novices, a GUI provides a quick and easy way to start using the tool. And even for advanced users, some software version management operations, like viewing branch history, lend themselves more readily to a GUI than to a command-line interface.

Perforce's GUI, P4V, is officially supported and available on all major platforms. P4V offers a nearly complete set of user functions, a growing subset of administrative commands, and visualization tools, including Stream Graph, Time-lapse View, the Revision Graph, and Folder Diff.

Other powerful applications include the Perforce Eclipse plug-in and a new Visual Studio integration. Perforce is leveraging HTML5, JavaScript, and web services to provide new integration and plug-in possibilities. Perforce's new Stream Graph, for instance, is built into P4V, and has already been replicated as a JavaScript-based applet.

Perforce also supports integrations with Microsoft Office and popular graphics programs like Adobe Photoshop.

Git is distributed with two GUIs, each offering a different set of features, and neither offering a featureset comparable to P4V. Several other, unofficial GUIs are available for Git, but are in various stages of completeness and stability. Git does not offer integration with Microsoft Office or programs like Photoshop. Thus, a Git user wanting to work in a GUI will need to learn more than one environment, and could still be forced to the command line for even non-administrative functions.

Identifying Changes

Perforce uses changelist numbers as unique identification points in the history of the entire repository. These changelist numbers are sequential in the order of commit. Each file also has its own revision numbers.

Besides serving as an identification point, pending changelists provide a way for developers to separate work into logical groups in a workspace. For instance, one changelist may include files for a bug fix, while another changelist contains files targeted at new feature work. Changelists also serve as the basis for Perforce shelves (see Code Sharing and Dependency Management).

Perforce provides two types of labels: automatic labels to reference a set of files at a particular revision, and static labels to provide additional flexibility. Both can reference subsets of files at different revisions or points in time.

Because Git must handle commits from different repositories, revisions receive a 40 digit SHA number as an identifier. Files do not have their own revision numbers. In order to determine the relationship between two revisions, a command such as `git log` or a graphical application must be used. The globally unique identifier is useful in some contexts but more difficult to work with for the typical user.

Git provides no facility for grouping work in progress in a single branch. By default `git commit` works on all pending files, and a developer can indicate individual files to commit. Separating work into logical units is often done by making private topic branches for each task.

Git tags are similar to Perforce automatic labels, but less flexible. They reference the state of the entire repository as of a particular commit.

Collaboration

TAnother primary function of software version management is to enable collaboration between team members. Collaboration may occur for teams working on a single project, between teams working on different versions of a single projects, or between teams working on distinct but dependent projects.

Code Sharing and Dependency Management

Often we may wish to share code between projects in the software version management system. For example, a common library module may be developed in its own project, but exposed as a read-only component to other projects.

Perforce's flexible data model and tools provide an easier solution for managing the relationships between different projects and modules.

Perforce Views and Streams

Perforce gives us several ways to handle this situation. If the other project is hosted in a different service, we can use Perforce's remote depot feature for code drops. For projects in the same server, we can branch a module into another project, and prevent modifications via the permissions system.

Perhaps the simplest approach uses flexible branch, workspace, and stream views to include part of one project as a component in another project. These views are easy to change, particularly when working with streams. The stream view, once defined by the project architect, flows automatically to all users of that stream.

This technique allows for selectively branching only a subset of a project, while still providing a fully populated working copy for local use. More importantly, it also provides an easy framework for managing dependencies between projects.

For example, consider a project with five modules. Of these five, three are actually imported from other products, and another is not going to be modified during the current development effort (see Figure 2).

It is important for the average user to easily obtain a coherent working copy of the project, with each module imported from the correct place, and with accidental modifications to read-only modules prevented. It is also important for the project architect, who understands the sometimes complex relationships between modules, to be able to easily define this information and share it with others. Perforce Streams provide this capability out of the box, where previously branch and workspace views with scripting assistance, were used.

Perforce labels are also very useful when working with modular digital development to reference disparate sets of files developed by different teams. These labels can then be used to identify the modules that comprise a release, for example.

Git Sub-modules and Sub-trees

In Git, each repository is a single project, so the only way to share code between two projects is to use either sub-modules or sub-trees. Sub-modules allow us to include one external repository as a sub-directory of another. The granularity of the sub-module concept is limited, however; we cannot expose just a piece of the external repository. This implies that defining the modules and their dependencies must be done correctly in advance, which can be difficult for complex projects.

Once a project is defined that includes one or more sub-modules, other developers can clone the project. Using sub-modules does require a bit of extra work, however:

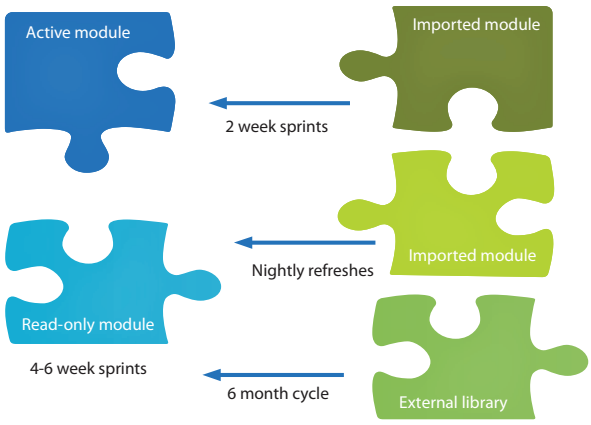


Figure 2: Modular development model

- The submodule `init` and submodule `update` commands must be used when cloning from a project that includes sub-modules.
- The submodule `update` command must be run separately from the `pull` command when updating a repository that includes sub-modules.
- When modifying a sub-module, it must be pushed separately from the rest of the repository.

Sub-trees are similar in concept to sub-modules; they allow the inclusion of one tree as a sub-directory in another tree. Sub-trees are normally used in read-only fashion, and require the use of a specific `git merge` strategy to work correctly. Similar to sub-modules, they require extra work to set up and maintain.

Consider again our project with five modules. Although Git would provide tools to assemble the five modules into a single working project, it requires extra initial setup and care when working. Git's tags (labels) would not be able to identify the cross-project set of files that comprise a coherent project. Perhaps more importantly, Git does not provide the access control mechanisms to prevent accidental updates of the read-only modules. Like many aspects of DVCS, code sharing requires careful planning and disciplined usage.

Branching and Release Management

The efficient management of branches is one of the core functions of the software version management system. Often called release management, this task includes designing an effective model for concurrent development, building and releasing a product, and maintaining releases. This section discusses the public aspects of branching and release management.

Project Creation

The process of creating a new project in the software version management system illustrates a fundamental difference between Perforce and Git. In Perforce, adding a new project simply means choosing an area of the depot to use for the new project, and deciding on an appropriate branch model. Once a depot location for the new project is chosen, the project manager defines a stream or template workspace and adds the new content.

In Git, a project is a repository. To add a new project in Git, we actually need to make a new repository.

We must choose the physical storage location for the new repository, and decide where the master repository is to be hosted for team use. Additional configuration may be necessary to support access via the HTTP or SSH protocols (see Security, Authentication, and Access Control). The repository deployment model must be designed and communicated to the project team. Then the new content can be added to the master repository.

Promoting Work

One common problem to consider is the need to merge upstream changes before promoting finished work.

This vital step ensures that the work in question has been reconciled against all upstream improvements and bug fixes.

With larger teams using Git, and branch operations done at the repository level, each user must be up-to-date against the entire upstream repository before promoting from his or her private repository. This leads to a work slowdown as the process of merging and promoting is effectively serialized.

Perforce, on the other hand, only requires that the individual files in question be up-to-date before promoting work. In practice, work should be merged and tested at the module level. This granularity allows easier concurrent work on large teams.

Understanding the Branch Model

A key part of release management is understanding the overall structure of the branch model. How does one branch relate to another, is the branch more stable, and where should changes in one branch be merged? Answering these questions is important when managing concurrent development by several teams, or maintaining older releases. Perforce Streams allow the project architect to codify this information.

Perforce uses the information to guide merges, ensuring that changes flow through the appropriate merge pathways. Streams provide an understanding of the structure of a project, and allow the project architect to define how change should flow. The intuitive picture provided by the Stream Graph also helps users understand the project, and most branching operations can be easily automated (see Figure 3).

Without Streams, a project architect could still implement an effective branch model using Perforce directory structure and workspaces.

Git does not offer a similar way to comprehend the overall branch model. This information must be understood by convention or documentation. Since a Git branching model may actually include several repositories, it is important that the release manager or project architect document the design.

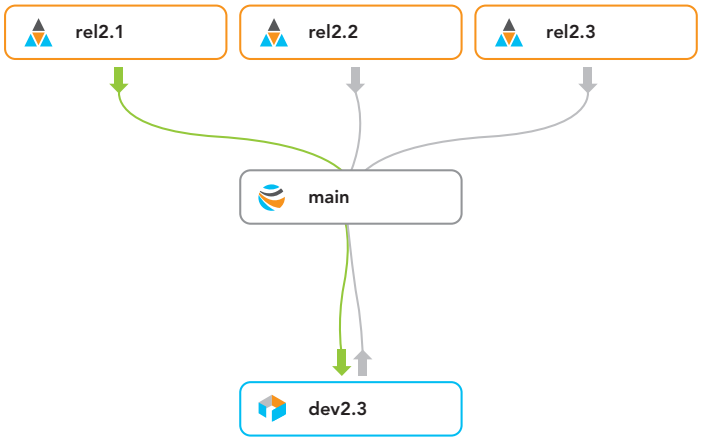


Figure 3: Stream Graph shows a branch model as defined by a project architect

A Release Management Scenario

Consider a project following the mainline branching model (see Figure 4). There are four branching levels: `dev`, `int`, `main`, and `rel`. The lowest level development branches focus on individual modules of the project. The diagram shows the typical branching operations that occur during development and release maintenance. Notably, work like bug fixes is merged down to less stable branches, and stable development is promoted up to more stable branches.

As discussed in other sections, the different release management and branching models in Perforce and Git have broad implications for ease of collaboration and administration.

The Scenario in Perforce

In Perforce, the release manager and project architect would define the branch model, and perhaps model it with a Stream depot. The *two dev* branches include only a subset of the project; this information is also included in the stream definitions. If at some point in the future the definition of the *dev* branches changes, the stream view is altered and that information flows automatically to all users.

Any user on the team can easily create a new workspace and start working on one of the development streams. Users may choose to use P4Sandbox for more effective private work, in which case P4Sandbox will handle the data transfer to the shared repository.

Access can be controlled for each level of the branch model using Perforce protections (see Security, Authentication, and Access Control). Write access at each level is granted to appropriate users, say only the project leads can integrate changes into the *int* branch. Access control changes are completed quickly using a graphical administration tool.

When changes are ready to merge, the Stream Graph provides visual notifications via the arrows between streams; merging changes to the right place is accomplished by simple point-and-click operations (see Figure 5).

Without using Streams, the process is essentially the same, but uses a different set of tools. For example, the MergeQuest component of the P4Eclipse plug-in offers a visual representation of a non-streams branch model.

The Scenario in Git

In Git, the release manager and project architect would define a multi-repository Git deployment (see Figure 6). A single Git repository could have the branches indicated in Figure 4, but without add-on tools, Git access control is at the repository level, not the branch level. For the purpose of controlling the flow of change, several repositories are necessary. Changes to the access control for the project may entail manually-intensive changes of HTTP or SSH access settings (see Security, Authentication, and Access Control).

This model, often called the "dictator and lieutenants" model, is commonly used in Git. Branches in a single repository are often short-lived, and having one person pull changes prevents other developers from running into merge problems.

Each developer has his or her own repository, representing the *dev* branches. Project leads (lieutenants) control the integration repository, representing the *int* branch. A project manager (dictator) would control access to the *main* repository and create release repositories as required. The lieutenants and dictator must pull changes from lower level repositories when necessary, or otherwise integrate patches from developers. This model also implies that, instead of switching branches to work on a bug fix for a release, a developer has to create a clone of the release repository.

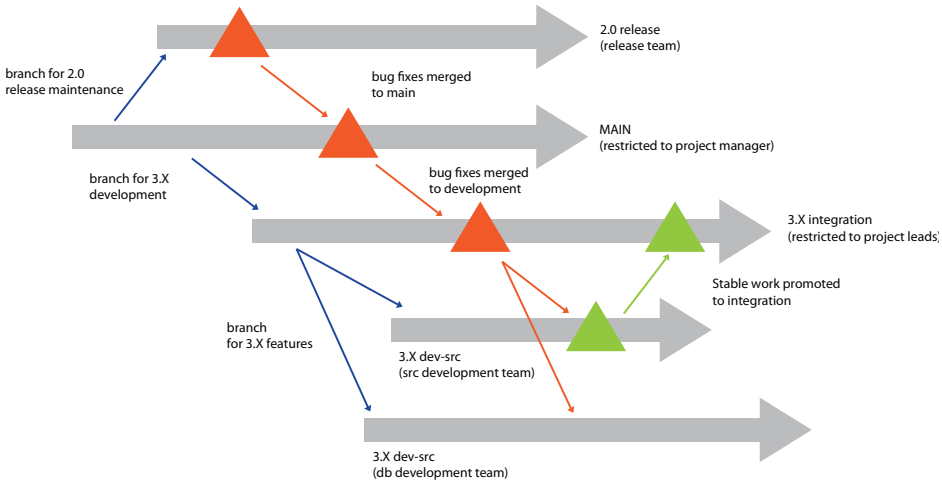


Figure 4: Mainline branch model

There is no single place to see whether any merges are pending between several different Git repositories. The Git `pull` and `push` commands can report pending merges between a parent and child repository, but do not show a complete picture of pending merges project-wide.

Since the *dev* branches need only a subset of the project, developers need to make sure to not modify the read-only modules in their project. Alternatively, the *src* and *db* modules could be developed independently, which would require the use of Git sub-trees or sub-modules and extra repositories (see Figure 7). In either case, the segregation of the project into modules must essentially be complete and static when development begins.

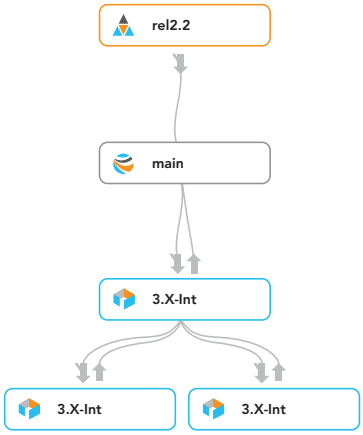


Figure 5: Branch model seen in Perforce Stream Graph

Digital Assets That Cannot Be Merged

Although enabling concurrent modification of files is a core function of a software version management system, some digital assets must be worked on serially. Some files are difficult, if not impossible, to merge, such as images, movies, and office documents. Other files are extremely sensitive to change, such as programming interface definitions. As a collaboration tool the software version management system helps prevent parallel work on these files.

Perforce has support for locking files in a single branch in order to prevent concurrent modifications. Locking can be accomplished on an ad-hoc basis or systematically using file type modifiers. And of course, Perforce can show if other users have a file checked out, which helps avoid potential conflicts. All of these benefits are made possible by Perforce's model of opening a file before starting work on it. Although Perforce users can choose to work without opening files first, in some cases the benefits of collaboration outweigh the cost of the extra workflow step.

Git's distributed design prevents any type of file locking or editing notification, so concurrent modification of files cannot be prevented.

Visual Tools for Collaboration

An important factor to consider when evaluating the effectiveness of a visual tool is the amount of information and context presented about the bigger picture. Knowing the overall branch model or the released projects for a single business unit is crucial information for many users.

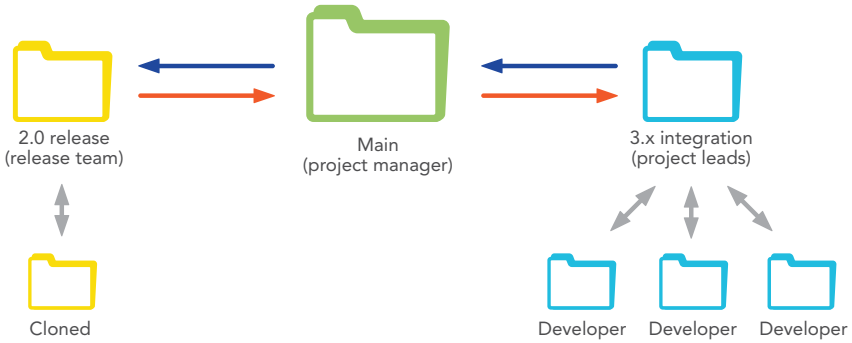


Figure 6: Repositories for branch levels

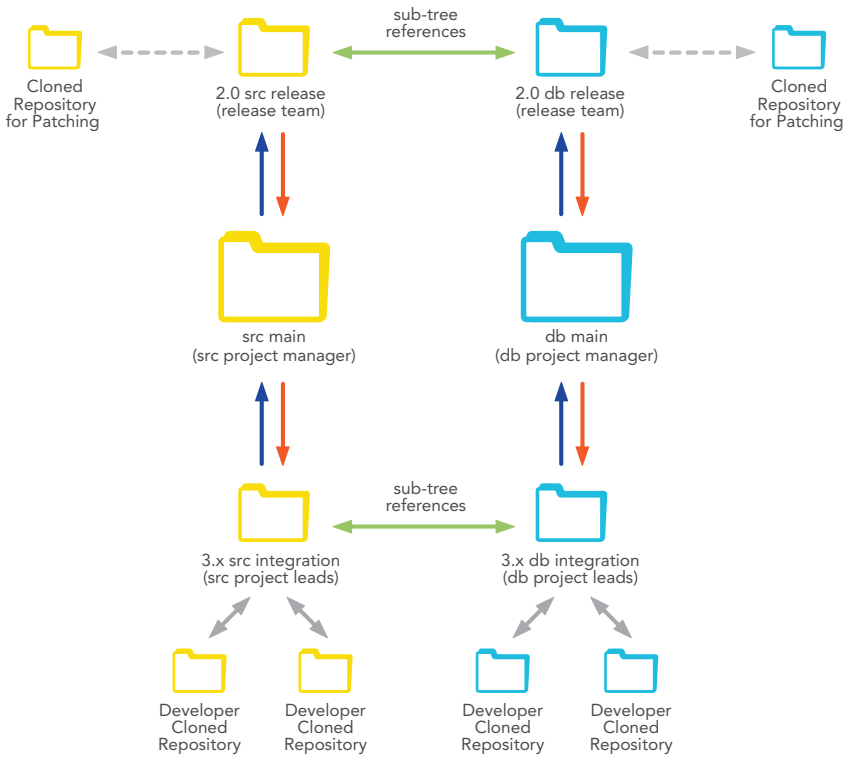


Figure 7: Git deployment using sub-trees

One Perforce *service* can host multiple projects and all of the important branches for those projects. Thus, the visual tools are able to show information about several projects, the branch model, and directory structure, giving the user a much richer visualization of the repository. These visual tools include the Stream Graph, Time Lapse View, Merge Quest, and Revision Graph.

Git graphical clients cannot show the relationship between a local repository and an upstream repository. Additionally, Git repositories are single projects, so a Git client can only show one project at a time. Without additional tooling, a Git user cannot know how his or her repository fits into the overall scheme of the project or team. The Git user must obtain this important information manually.

Application Lifecycle Management (ALM) and Community Development

ALM describes both a process and the associated tools for managing the evolution of a project from requirements through design, coding, test, and release.

A software version management system is the foundation of successful ALM, providing the basis for code review, continuous integration, and traceability of requirements and defects.

Part of the appeal of modern DVCSs systems like Git is the availability of popular community sites such as GitHub. These sites offer ALM-like features, such as discussion forums, basic defect tracking, traceability, and notifications. Many of these sites also provide additional management tools for security and repository administration.

It is difficult to design one ALM toolset that meets the needs of all users. Perforce offers superior ability to integrate into any ALM or community site, by virtue of its fully-supported range of APIs and the upcoming web services framework. Since Perforce supplies its own comprehensive set of security and administration tools, it only relies on other tools to provide complementary features.

The various community sites are perhaps best suited for projects with very light collaboration. For true collaboration in a team setting, Perforce has superior collaboration tools that make it easier to accept contributions from others, fostering a more dynamic interaction.

Perforce provides free licenses for open source projects, so a Perforce service can act as the foundation for a public community site. Perforce's own upcoming community development site will serve as such an example.

SCALABILITY

Software version management provides the foundation for modern digital asset development. A software version management system must support a large user base distributed around the world, and accommodate heavy usage.

Large File Management

Many industries use a software version management system to manage large digital assets, including multimedia, graphics, and documentation. These files, which are typically binary rather than textual, can be very large.

Perforce is efficient at storing and distributing files of any type or size. It has features that allow the storage of only a small number of revisions of files, saving space on the service. Developers can, using their workspace view, choose not to copy binary files from the service. Legacy data can be moved into archive (offline) storage areas.

Git, on the other hand, is inefficient at managing binary files:

- Since the entire repository is stored in every workspace, storing large numbers of binary files leads to bloated workspaces. Git has no way of only storing a handful of revisions of these files. (Git's sparse checkout feature does allow a working directory to exclude certain files, but the data is still in the `.git` directory.)
- Git works at the repository level, and must process every file in the repository for some actions, calculating a SHA1 ID for each file along the way. Repositories over a few GB in size can quickly become difficult to use on an average workstation.
- Git has difficulty adding large binary files to a repository, because it uses a large amount of memory to process new files. For example, adding a 3GB file on a workstation with 4GB of RAM often results in out of memory errors.

Git is designed to work with text files, and the Git community recommends using a dedicated system to version build artifacts and other large files. That implies a second system with additional overhead and the challenge of linking derived assets to the source files in the software version management system.

Cross-Platform Compatibility

For enterprise users, the software version management system should be well supported on all platforms in use. Installation and upgrade procedures must be seamless and well documented.

The Perforce service program is available on Windows, Mac, and most major versions of UNIX/Linux. The Perforce application

programs are even more widely available. Perforce features full interoperability between services and applications on different platforms. The various Perforce APIs (including the command line application) are supported across all major platforms, an important point for software version management system administrators and power users who rely on scripting.

Git, on the other hand, is only available as a reference implementation on Linux; and more specifically, the only binary distribution available from the official site is for the RPM package manager. Well-supported binary distributions are available from other sources for Mac OS X, Solaris, and other flavors of Linux and UNIX.

Windows support for Git originally relied on the Cygwin environment. A native Windows distribution supported by the community is now available. Setting up a shared Git repository on Windows still relies heavily on the Cygwin environment to provide support for SSH and `git-daemon`.

Remote Development

A software version management system must support teams of users at geographically remote locations. Remote development support implies supporting the collaboration between these teams, ensuring good performance globally, and supporting users who are working without a connection to a shared network.

Git, as a DVCS, is well suited to distributed development scenarios. It requires no sustained network connection between distributed users, and also offers commands like patch and bundle that support very sporadic collaboration. How effectively Git can be used to sustain collaboration on a large team is an important question addressed in the Collaboration section.

Perforce as a shared repository has a powerful set of tools to support remote teams and maintain performance (see Figure 1). These tools include the Perforce proxy, broker, and replication services. Remote depots also facilitate code sharing between separate service. In the future, support for federated service architecture will improve remote support at the shared repository level.

P4Sandbox offers excellent support for users without a sustained network connection to the shared repository. Similar to Git, it offers enough information to allow independent work.

Extensions: APIs and Scripting

Release managers, administrators, and other power users often want or need to extend the software version management system. Custom integrations with other tools, build processes, scripts that assist users and enforce policy, and backup scripts are just a few examples of the useful ways to build on the software version management system. How easily these extensions can be implemented is an important aspect of the software version management system.

Perforce has a command line application and several well-supported APIs, including the C/C++ API, several APIs for scripting languages, and a web services framework. Since all of these elements are officially supported, users can rely on consistent functionality on different platforms and using different programming languages.

Git has an official C/C++ API, and several unofficial contributed APIs for other languages. Since most of the APIs are not officially supported, they may not be updated frequently, or may become obsolete entirely

if community support wanes. A user requiring a stable, long lived set of scripts and tools will be best served by using the C/C++ API or the Git command line application.

ADMINISTRATION AND MANAGEMENT

Because the software version management system contains valuable data and is the foundation for most digital asset development activities, it must be managed effectively. Backups, security, and other administrative tasks should be reliable and easy to manage.

Setup and Deployment

The Perforce shared versioning service architecture is simple. For core version management system functionality, the service software is self-contained, including an embedded database for tracking the metadata. By using standard file systems, network features, and a single service, the only overhead added by Perforce is that associated with backups and regular user maintenance activities. The release management strategy has little impact on the administrators.

Typically, one part-time Perforce administrator is adequate for supporting about a hundred users.

Very little daily involvement from IT is necessary after the service is installed; the Perforce administrator can manage users, groups, and access control from within the product. (There is support for authentication or single sign-on via LDAP or Active Directory if required.)

Administering a single Git repository is simple. However, since Git relies on host SSH or HTTP accounts for authentication to a shared repository, IT will be more heavily involved at many sites. IT will also need to be involved in choosing and configuring the protocol used for communication between repositories. Perhaps most importantly, the Git deployment is heavily influenced by the release management strategy (see A Release Management Scenario).

Security, Authentication, and Access Control

Perforce and Git operate under very different security and authentication models.

Perforce supports a granular security model based on users and groups. Authentication control is managed using passwords, with the option to enforce password strength, or by integration with an external password checker. Access control is centrally administered through Perforce's protection table. Access control allows for several levels of access, and can be applied at granularity down to individual files. Such a system is designed for strong security management at organizations of any size, and allows access to valuable intellectual property to be tightly controlled.

Git, on the other hand, was designed for a completely distributed environment, presumably with each user on an isolated workstation. Authentication control for a local repository is not an issue, because users are on different networks; thus, Git does not have the equivalent of a login command, and has no central concept of users and groups. Rather, each user configures Git with his or her (self-assigned) user name and email address.

Access control for Git shared repositories is managed by granting *pull* (read) and *push* (write) access. Granting access is done by creating SSH or HTTP server accounts for the committers, and hence is not granular, in terms of level of access or area of access. Since granting access to a shared repository can involve granting write access to the server's file system via SSH, a security breach can corrupt the physical server itself, rather than just the software version management system data.

Additional products exist that provide easier and more granular access control for Git. However, these products are not part of the official Git distribution, and often rely on Cygwin in Windows environments. Hooks can also be used to provide more granular access control, but such a solution is ad-hoc and requires heavy customization.

It becomes apparent that in all but the simplest environments, Git's security model is insufficient. Without additional tools or products, managing security for a Git deployment is complex and time consuming.

The lack of security facilities may actually be an advantage for small, distributed open-source communities, since it implies less administration, and users presumably know and trust each other. However, the model would not scale well as open source projects become larger. The inability to grant commit privileges to only small portions of a large project is a significant disadvantage.

These scalability problems are the driving force behind Git add-on layers such as Gitosis and Gitolite, which attempt to provide solutions to the authentication and access control problems.

Additionally, in large environments, the need to set up a new repository for each project (or long-lived branch level) is a daunting prospect, implying additional overhead for security, backups, and other tasks.

Auditing

In many environments, security concerns dictate strong auditing controls for a software version management system. Perforce provides an audit log mechanism that can record which users accessed parts of the repository. Git does not provide a similar facility; any authorized user can pull changes from a repository with no audit trail, unless a logging system from another tool is used.

Backup and Maintenance

The importance of a rigorous backup and recovery plan is independent of the type of software version management system used. However, fully backing up a Perforce service is easier to manage in enterprise settings.

The Basics

Perforce backup and recovery procedures are well documented and easy to implement. The tools for managing a warm or nearly hot spare are reaching maturity, including full repository replication tools. A backup of a Perforce repository normally includes important supplementary data such as triggers and the protections table (access control settings).

Backing up a Git server is very simple: clone or copy the repository to a new server. Additional procedures are necessary to back up hooks and security settings, which may depend on the host operating system, a web server, or other user management software.

Since a Git deployment may consist of several repositories, the backup and recovery plan must include all of the important repositories.

Repository Verification and Maintenance

Both Perforce and Git provide tools to verify repository integrity and optimize the metadata. However, Perforce's tools are more consistent and easier to deploy in a team setting.

Perforce's tools can easily be run as part of a backup strategy. If any corruption is discovered, there are well-documented recovery strategies for common problems. In a Perforce deployment, these tasks are administrative in nature and transparent to users.

Git repositories should be frequently packed to ensure performance and efficiency. Git has a data integrity checker but recovery procedures are not consistent and may involve recovery from clones of the repository. These chores are a responsibility for each user.

Recovering Lost Workspaces

Application workspace metadata is stored on the Perforce shared versioning service, and can even be versioned using a spec depot. If a workspace is lost due to disk failure on a workstation, the definition of the workspace—including metadata that records which file revisions are synced to the workspace—can be recovered. The only data lost would be any file content changes not yet submitted to the versioning service.

In Git, the `.git` directory contains the repository/workspace metadata. If this directory is lost or corrupted, then the workspace is essentially unusable. If a workspace is lost, there is no point of recovery unless a local backup procedure is in place.

In practice, this distinction is most important at the point at which a workspace definition contains valuable information. For a shared Perforce Service, versioning

the definition of a workspace is practical and prudent. For a Git repository that is serving as a collaboration point, such as a repository used for integration testing, the repository would need to be regularly backed up.

Policy Management

Many sites choose to enforce business policies in the software version management system. For example, a commit policy may require changes in an important interface to be documented in the same commit. Policy management can provide important structure and guidance to the team.

Triggers

Triggers are commonly used to manage policy in the software version management system.

Perforce manages triggers centrally on the Perforce service. Trigger configuration is versioned in the trigger table using the spec depot, and with some simple techniques the triggers themselves can be version managed. A wide variety of trigger types allow administrators to easily enforce policy or provide guidance to users. By working in P4Sandbox, individual developers can work outside of the scope of these policies, but still be subject to the policies when sharing changes with the shared versioning service.

Since Git does not have a shared server, hooks must be installed on each clone of each repository. Hook configuration is not versioned, and there is no easy facility to map a version-controlled hook into the appropriate directory. Although hooks can be used on the master repository to enforce policy or provide guidance to developers, there is no central way to manage these hooks for all repositories in a project. Furthermore, git commit has a no-verify option that lets users bypass some of the commit hooks. Effectively managing Git hooks would require an additional layer of tools.

Additional Options

Perforce provides other options to manage software version management system policy. The Perforce Broker can act as an intermediary between users and the shared versioning service. It can reject actions under some conditions, redirect commands to other service, and provide new commands and functionality. For example, the Perforce Broker can add new commands to create a new project. These commands would handle the mechanics of branching, access control, and stream definition.

Other than hooks, Git does not offer a standard framework for ad-hoc policy enforcement. New commands can be added by writing plug-ins, but there is no equivalent to the Perforce Broker.

Many sites use a Git workflow where policy is enforced by upstream committers, or their customized hooks. (See A Release Management Scenario.)

Reporting and Data Mining

The information in the software version management system metadata provides important metrics for the product managers. This information can answer key questions about how many files are changing during some timeframe, or how sensitive a particular module is to bug reports.

Both Perforce and Git provide command-line tools and API support for data mining. However, Perforce's shared repository will provide a unified view of an entire project (see Understanding the Branch Model). A single Git repository will provide information on only a single project, and probably only a single important branch of a single project.

For intensive data mining, the Perforce database can be replicated into a relational database. Report engines can then be used to run custom reports.

Support and Services

Perforce Software provides technical support and related services, including training, e-Learning, consulting, and remote administration. Technical support is available around the world on a 24x7 basis.

Git is informally supported by the user community. Support and services may be available from third party vendors.

CONCLUSION

The challenges of modern digital asset development continue to grow. Some of these challenges affect users at the private or individual level, where users want to work creatively and quickly in a resource-constrained environment. DVCS has proven very effective at addressing these challenges.

Other challenges impact the team or enterprise. Pulling together the work of a team of individuals into released products, scaling up to support complex development efforts and large teams, and managing security are all important considerations. Perforce has excelled at managing the needs of teams and enterprises.

Git could be combined with Perforce or other systems

to provide the collaboration, scalability, and management tools necessary in a complex environment. However, introducing a second version management system adds more complexity.

With P4Sandbox, Perforce is offering a unique solution to the problems faced by individual developers. Like Git, P4Sandbox allows for productive private work. But P4Sandbox also offers easy collaboration and access to the power of the shared repository, as it works seamlessly with the main Perforce shared versioning service. This solution meets the needs of all users of the software version management system—individuals, teams, release managers, and administrators—with a unified, powerful set of products.

LEARN MORE

Evaluating Perforce

More than 400,000 users at 5,500 companies rely on Perforce for enterprise version management. Perforce encourages prospective customers to judge for themselves during a typical 45-day trial evaluation. Free technical support is included with your evaluation. Get started: perforce.com/trial

Scheduling a Demo of Perforce

To learn more about Perforce, schedule an interactive demo tailored to your requirements:

perforce.com/product/demos

Migrating to Perforce

Perforce Consulting Services has experience assisting customers with migrations from various legacy software version management systems. For more information, visit: perforce.com/consulting

perforce.com

North America
Perforce Software Inc.
2320 Blanding Ave
Alameda, CA 94501
USA
Phone: +1 510.864.7400
info@perforce.com

Europe
Perforce Software UK Ltd.
West Forest Gate
Wellington Road
Wokingham
Berkshire RG40 2AT
UK
Phone: +44 (0) 845 345 0116
uk@perforce.com

Australia
Perforce Software Pty. Ltd.
Suite 3, Level 10
221 Miller Street
North Sydney
NSW 2060
AUSTRALIA
Phone: +61 (0)2 8912-4600
au@perforce.com