Introducing Perforce Streams

Laura Wingerd, Principal Consultant, Perforce Software

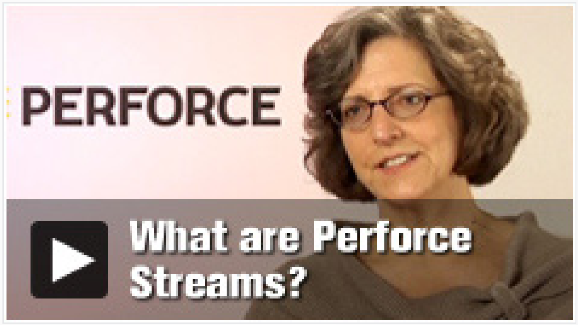
This ebook will give you a quick introduction to Streams and a look at the big picture: why streams were developed and what problems they solve. Then we'll move on to a quick tutorial to make things more concrete. At that point we'll be ready to look at a few big concepts in more detail.

Just a note: I won't be covering the Streams visual tools much in this ebook. If you'd like to see those in action, you can view our Streams webinar and a short video, or install a trial.

PERFORCE
Version everything.

About the Author

Laura Wingerd recently retired from her position as Vice President of Product Technology for Perforce Software. During her long tenure, Wingerd helped guide the evolution of Perforce's version management platform and architected their most recent innovation, Perforce Streams. She now splits her time between retirement and consulting. She is also the author of "Practical Perforce."



This ebook was developed from a series of blog posts from the Perforce blog, p4blog.

The Ultra-Quick Streams Introduction

Let's start with a few quick questions to whet your appetite...

What Are Streams?

In the world of version control, a branch is a set of files that is a variant of another set of files. A stream is a branch with brains.

For example, say my branch is the set of files in `//depot/sandbox/laura/ourApp/` (this is an example of a typical repository path in Perforce). You could copy my files into `//depot/sandbox/bob/ourApp/` and that would be your branch. Our branches can evolve independently, and we can merge changes between the two.

Perforce has tools to compare and merge our branches, and it's very good at keeping track of what we've merged. But Perforce doesn't actually know what the scope of each branch is. Jane, for example, could branch all of `//depot/sandbox/` into `//depot/jane/sandbox/`, cloning both my branch and your branch into her branch. That's probably not what Jane meant to do. What she wanted was to clone my branch, or yours, but surely not both.

"In the world of version control, a *branch* is a set of files that is a variant of another set of files. A *stream* is a branch with brains."

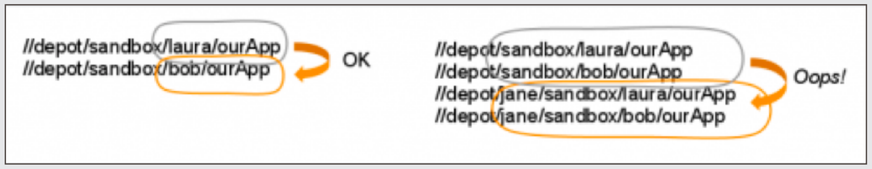

//depot/sandbox/laura/ourApp
//depot/sandbox/bob/ourApp
OK
//depot/sandbox/laura/ourApp
//depot/sandbox/bob/ourApp
//depot/jane/sandbox/laura/ourApp
//depot/jane/sandbox/bob/ourApp
Oops!

Figure 1: Mistaken branch creation

Until now, however, Perforce hasn't had the metadata to guide Jane toward what she really wanted. It was up to Jane to know in advance what part of the depot (repository) held the branch she wanted to clone. And if she got it wrong, she could end up cloning the wrong files. Since Perforce didn't know where our branches were, it couldn't show Jane which files to clone to make her own branch.

Enter streams: A Perforce database object called a "stream" now describes information about a branch—its location in the depot, its owner, its parent branch, and more (upon which I will elaborate soon enough). Jane can now clone the right set of files simply by telling Perforce which stream to start from. And now Perforce can guide Jane toward integrating (merging) changes properly, because it know how her branch relates to the other branches.

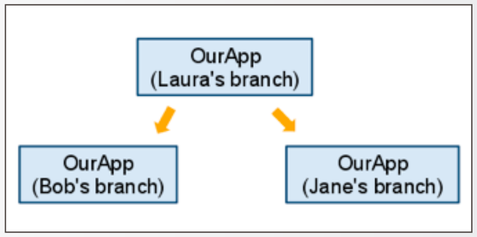


Figure 2: Branches with brains

Doesn't Perforce Have "Branches" Already?

I hear you saying to yourself, "Wait a minute—isn't that what the `p4 branches` command is for?" True, there is a `p4 branches` command in Perforce, and yes, you can create a Perforce object called a "branch." But if you've ever tried it, you'll realize that what Perforce is calling a "branch" is actually a mapping between two sets of files.

"Branch" is a tricky term for Perforce. Because we already use it to describe a mapping, it gets confusing when we use it to describe a single, unilateral branch. (The Perforce manuals avoid confusion by using "codeline" to refer to a unilateral branch.)

So don't trip on the terminology—the existing Perforce "branch" object doesn't represent a unilateral branch of files. That's what the new "stream" object will do.

Did Perforce Invent Streams?

No. Streams have been around for a long time. I first heard the expression from Brad Appleton in his Streamed Lines paper. I ran across it again in Brian White's ClearCase book. Today, a decade later, a number of version control products feature streams.

What Perforce did do, however, was combine some of its own most distinctive features with the convenience and appeal of streams.

Why Perforce Streams?

Many shops are already using Perforce for branching strategies that are very similar to what we're offering with Streams. In fact, the stream model we've designed comes from the ideas our users have shared with us. (I'd name names here, but I'm too afraid I'll leave someone out. All I'm going to say is that many of you—and you know who you are—have contributed to the vision.)

So, if you can already do perfectly good branching with Perforce, how do streams add value? In these ways:

- ***Streams offer an out-of-the-box branching and merging strategy.*** Instead of spending your time figuring out your own strategy, you now have the option of using ours. This is great if you're new to Perforce, because you can get started faster. If you're already using Perforce, not to worry—our stream model is designed to work shoulder-to-shoulder with "classic" Perforce.
- ***Streams are effective.*** The Perforce stream model is based on concepts we've promoted a lot. (See Chapter 7 of my book, for example, or my Google talk.) The feedback we get is that yes, these concepts work. The mainline model, the tofu scale, and the merge-down/copy-up protocol—all of these make branching and merging less complicated and more agile.
- ***Streams give control to project leads.*** A stream's view determines which depot files are in the stream and how they will appear in users' workspaces. The owner of a stream controls its view. And stream views are inherited, so that when a stream is created or reparented, its view automatically reflects its parent's view.

"So, if you can already do perfectly good branching with Perforce, how do streams add value?"

- ***Streams are built on Perforce's most powerful features.*** Take client view mapping, for example. Files don't actually have to be branched to appear in a stream. Instead, they can be "imported" from the parent stream or from other streams in the system. (I use quotes here, because "import" is just a word we're using to describe the effect. The underlying mechanism is automatic client view mapping.)
- ***Streams are easy.*** With streams you'll never have to edit another client or branch view. For project leads, this means no more coaching individual developers to set up their workspaces—just set up one stream view and all workspaces for that stream will have the proper view.
- ***Streams are pretty.*** You are going to love our stream visualization tools!

"With streams you'll never have to edit another client or branch view."

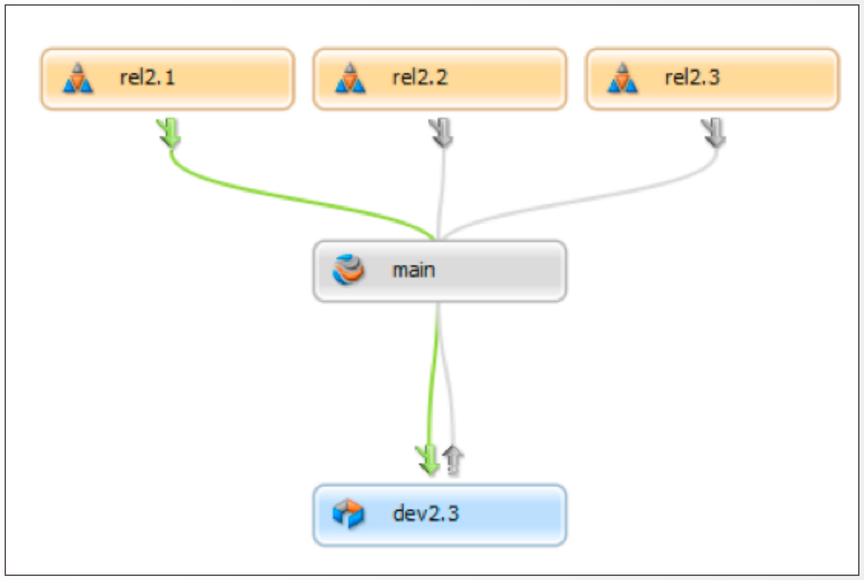


Figure 3: The Stream Graph

What's Under the Hood?

The Stream feature is essentially a branching and merging application based on the Perforce system. Underneath it all, you're still working with depot files, client workspaces, and all the usual Perforce stuff—none of that changes with Streams.

What's new is that two pillars of Perforce, the Perforce Server and P4V, are doing a lot of housekeeping to make using streams easy. The server, for example, now generates client views and branch views dynamically from stream definitions. It also uses stream lineage to compute sensible defaults for merges and promotions. And P4V is being enhanced to show streams in a slick visual flowchart, and to offer a simple interface for sophisticated stream tasks. Other Perforce clients and integrations will support streams, of course.

"Until now, Perforce users have relied on externally defined road maps for wrangling codelines. Perforce Streams now offers that capability out of the box."

The Big Picture

Now let's jump into the big picture. Until now, Perforce users have relied on externally defined roadmaps for wrangling codelines. Perforce Streams now offers that capability out of the box. In this section I'd like to paint the broad, high-level picture of streams. For now, I'll just touch on key points, or the 10,000-foot view. As we'll see, the birds-eye view of a project is one of the most powerful aspects of streams.

The raison d' être of streams is to model branching and merging intent. Streams do that through their parent-child relationships. So one way to look at streams is with a flow diagram:

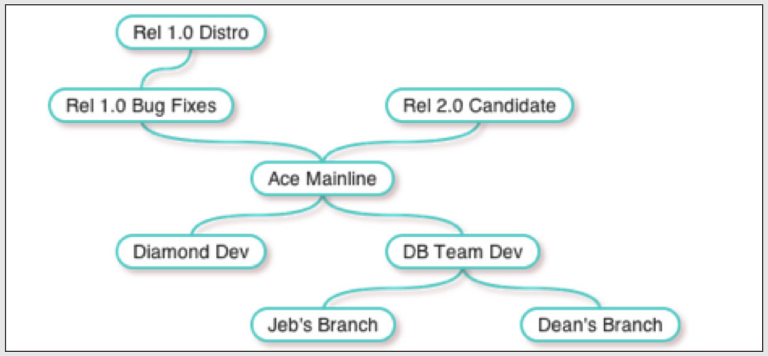


Figure 4: A streams flow diagram

What you see here is a mainline (trunk) and the streams in its lineage. Changes are intended to flow along the connectors between streams. The intended flow of change can be changed by changing the stream lineage. I say “intended flow” because it still takes a user action to branch and merge files. However, the Perforce commands for branching and merging are stream-aware, with defaults that support intended flow.

The Flow of Change

Laura Wingerd
December 4, 2006

In the Perforce model, streams have what we call “stability.” Development streams are less stable than their parents; their role is to keep the churn of active development from destabilizing code that is or will soon be released. Release streams are more stable than their parents; they assure that the exacting tasks of the release pipeline don’t hold up active development. In the middle is the mainline, which provides ultimate control over all the streams related to it.

In the diagram, the more stable stream is always shown above its less stable parent or children. This tells two stories. First, it tells us that the higher up a stream appears in the diagram, the riskier it is to do work in it. Second, it tells us how to propagate change along the stream connectors: merge down, copy up.

So that’s the model. (Does it sound familiar? You heard it in The Flow of Change.) Now, let’s talk implementation.

Stream specifications are stored by the Perforce Server, just like specs for workspaces, users, groups, depots, and other Perforce objects. A stream spec controls files in a codeline—in fact, a stream's unique ID is the depot location of the files it controls. Stream identifiers are immutable, and because of that, streams also have owner-modifiable names. Here's an example of a stream spec:

```
Stream:       //Ace/DEV
Parent:       //Ace/MAIN
Owner: jack
Name:  Diamond Dev
Type:  development
Description:
   Shared development stream for Diamond
features.
Paths:
        share ...
```

The stream spec also controls two internal views. One view defines the scope of workspaces (we've been calling this the "logical view" of the stream), and one view is used by commands that compare and integrate files. A stream's owner can customize the template from which the views are generated. For small streams, the default template is not likely to need attention. For large streams, where enormous file trees are involved, the template can maximize code reuse and enforce best practices. (More on that later.)

All streams descending from the same mainline are rooted in the same depot. The streams `//Ace/MAIN`, `//Ace/DEV`, and `//Ace/jebwork`, for example, are all rooted in the Ace depot. A stream's root is always one level below its depot's root, making a flat and predictable namespace. You can root as many streams as you want beneath the same depot, but we recommend having only one mainline per depot. Stream depots are for the most part just like regular Perforce depots. Another way of looking at streams is as subdirectories in a depot tree:

"If you switch a workspace from one stream to another, or if you modify a stream's view template, your workspace view changes on the fly."

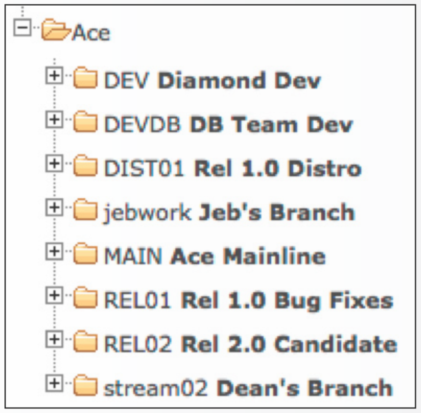

Ace
DEV **Diamond Dev**
DEVDB **DB Team Dev**
DIST01 **Rel 1.0 Distro**
jebwork **Jeb's Branch**
MAIN **Ace Mainline**
REL01 **Rel 1.0 Bug Fixes**
REL02 **Rel 2.0 Candidate**
stream02 **Dean's Branch**

Figure 5: Streams in the repository

Here's what's different about stream depots: You must use a stream workspace to submit files to them. A stream workspace "belongs" to a single stream, and gives you the logical view of that stream. With stream workspaces you don't have to set up a client view; views are automatic. If you switch a workspace from one stream to another, or if you modify a stream's view template, your workspace view changes on the fly.

So that's the big picture: Streams model your project. They describe what codelines are in a project, how they relate to one another, and how change flows between them. Streams have a very predictable storage model in the server, and that predictability lets the server generate workspace and child stream views automatically.

A Tiny Tutorial

Now that we've seen the big picture, let's dive right in to a tiny tutorial and see Streams in action. Here's the short synopsis of what we'll do:

- Create a stream depot.
- Create a mainline rooted in that depot.
- Import files from an existing depot into our mainline.
- Branch a development stream from the mainline.
- Merge files from the mainline to the dev stream.
- Promote the dev stream to the mainline.

Just a word of caution—the commands and syntax you see below may change over time. Be sure to check out the documentation and command reference to double check the details.

There will be visual tools for Streams, but I'm going to use the command line for this tutorial. And I'm not going to dwell on syntax or command arguments at this point—there's plenty of time for that later.

Let's assume our local environment is already conf igured to connect to a stream-savvy Perforce Server. Before we get started, we need to bootstrap a new client workspace. My trick for this is to create a new local directory, and in that directory, run:

```
% p4 workspace -o | p4 workspace -i
```

Note that I don't bother changing the view at this point.

Okay, let's roll!

Creating a Stream Depot

Streams live in stream depots, so that's the first thing we need. Let's define one called Ace:

```
% p4 depot Ace
```

We edit the default definition to make this a stream depot:

```
Depot: Ace
Type:  localstream
```

(Note that the "we" in this step is a user with super privileges—a Perforce administrator. But from here on out, nonprivileged users can run the commands I'll demonstrate.)

Creating a New Mainline

Now we define a stream called //Ace/MAIN:

```
% p4 stream -t mainline //Ace/MAIN
```

I'll leave the default settings alone for this stream.

Importing Files into the Mainline

Having defined the stream, we can now switch our workspace to it:

```
% p4 workspace -s -S //Ace/MAIN
```

Our client view is now a view of the `//Ace/MAIN` stream. (You can also use this version of the "workspace" command to change your workspace view based on a workspace template.) There aren't any files in that stream yet, so let's branch some from another depot location:

```
% p4 copy -v //depot/projects/ace/trunk/... //Ace/MAIN/...
% p4 submit -d "Seeding the Ace mainline"
```

Now our mainline is full of files, ready for us and our colleagues to start working on them. Anyone who wants to work in the mainline can create their own workspace and switch it to the stream, just as we just did. Then they can run the usual p4 sync, p4 edit, and p4 submit commands, the same as in a nonstream workspace.

Branching a Development Stream from the Mainline

Oh, but wait—let's say we don't want to do feature development work in the mainline. Instead, we want to work in a development stream. So let's branch a development stream from the mainline. We'll call it //Ace/DEV. First we define it, with //Ace/MAIN as its parent:

```
% p4 stream -t development -P //Ace/MAIN //Ace/DEV
```

We now switch our workspace to //Ace/DEV:

```
% p4 workspace -s -S //Ace/DEV
```

And we populate the stream by branching its parent's files:

```
% p4 merge -S //Ace/DEV -r
% p4 submit -d "Branching from mainline"
```

This leaves our local workspace in sync with the `//Ace/DEV` branch, ready for us to start working on files.

Merging Files from the Mainline

While we're working on features in `//Ace/DEV`, other changes are being submitted to `//Ace/MAIN`. Here's how we merge those changes into the `//Ace/DEV` branch:

```
% p4 merge -S //Ace/DEV -r
% p4 resolve
% p4 submit -d "Merged latest changes"
```

(Yes, **p4 merge** is a new command. It does pretty much the same thing **p4 integ** does, but only after validating the relationship between a stream and its parent.)

Promoting Our Development Work to the Mainline

Let's say we've completed a development task in the `//Ace/DEV` stream. Now we'd like to promote our work to the parent stream.

"Promote" is simply another way of saying "copy up after merging everything down." So let's make sure we've merged everything down first:

```
% p4 merge -n -S //Ace/DEV -r
All revisions already integrated.
```

Looks good. Now let's switch our workspace back to `//Ace/MAIN`:

```
% p4 workspace -s -S //Ace/MAIN
% p4 update
```

We run **p4 update** after switching the workspace, because both streams have files in them at this point. (You'll be happy to know that **p4 update** is smart enough to swap out only the files that aren't the same in both streams.) And that, folks, is simple in-place branch switching.

Finally, we copy content from the `//Ace/DEV` stream to its parent:

```
% p4 copy -S //Ace/DEV
% p4 submit -d "Here's our new feature"
```

Et voila—our work in the `//Ace/DEV` stream has just been promoted to `//Ace/MAIN`.

This tutorial is just the tip of the iceberg. In the next sections we'll look at controlling the flow of change with streams, and then how easy it is to control stream and workspace content with stream views.

Streams and the Flow of Change

Now let's look at how stream types and stream behaviors can orchestrate the flow of change: where and how changes are merged.

Stream Types

There are three stream types in Perforce: “mainline,” “development,” and “release.” A mainline is a stream with no parent. Its stream spec might look like this:

```
Stream: //Ace/main
Parent: none
Type:   mainline
```

(Every stream is defined with a spec. I’m showing only the relevant portion of stream specs in these examples.)

Only one type of stream can be an orphan, and that’s a mainline. A stream with a parent must be either a development stream or a release stream. The distinction is in how change is expected to flow—by merging, or by copying—between a stream and its parent.

Merge Down, Copy Up

For example, consider this stream:

```
Stream: //Ace/laura-dev
Parent: //Ace/main
Type:   development
```

The `//Ace/laura-dev` stream is a child of `//Ace/main`, and the fact that its type is “development” tells us that Perforce expects it to pull changes from `//Ace/main` by merging, and to push content to `//Ace/main` by copying.

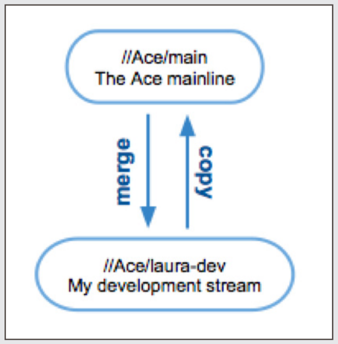


Figure 6: Merge down, copy up

This is very much how branching works in other version control systems. You could think of development streams as normal branches, used by normal developers, for normal tasks. For a small project, a mainline and a handful of development streams may be all you need.

A "release" stream is the opposite: It pulls from its parent by copying, and pushes to its parent by merging. This may sound odd to you if you haven't read my book or seen any of the Flow of Change presentations we've given. Consider this use case:

```
Stream: //Ace/stable
Parent: //Ace/main
Type:   release
```

Here we have `//Ace/stable`, a release stream whose parent is the mainline. We've branched it because we need a place to test and stabilize our next release. At the same time, we don't want to keep developers from promoting their work on future releases into the mainline. We do our system testing and bug fixing in the release stream, and as we fix bugs, we merge the fixes into the mainline.

And let's say our product is a web-hosted product. When the `//Ace/stable` stream is ready to ship, we label it and publish it to our live content distribution site. This frees the `//Ace/stable` stream for stabilizing the next release candidate, which we promote (copy, that is) from `//Ace/main`, and so on.

Unlike development streams, which are typically less stable than their parents, a release stream is more stable than its parent. A convention I started using in my book was to show stability on a vertical scale, from low to high. The flow between `//Ace/stable` and `//Ace/main`, then, looks like Figure 7.

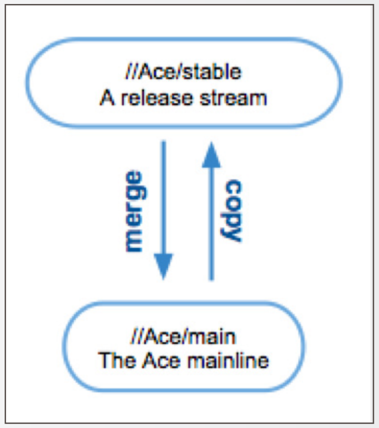


Figure 7: Release streams merge to parent

Just think "merge down, copy up." If you'll take a look back at Figure 3 and Figure 4, you'll see diagrams showing a mix of release streams and development streams. The release streams are high on the diagram, and the development streams are low. As you can see, "merge down, copy up" applies to all the streams in this system.

(An aside: I used to talk of streams being “firmer” or “softer” than their parents, but outside of the people who’d read my book, no one knew what I meant. Now I say “more stable” and “less stable,” though that’s kind of a mouthful.)

Merge and Copy Commands

So, a release stream expects promotions from its parent, and a development stream expects merges from its parent. What do we mean by “expects”? We mean the way the new copy and merge commands work.

With the advent of Streams, many Perforce commands now sport the -S flag, which tells them to operate on a stream. The merge and copy commands are special. Not only do they allow the `-S` flag, they check the expected flow of the stream with respect to its parent. For example:

```
p4 merge -S //Ace/stable
```

This command merges changes from `//Ace/stable` to its parent, `//Ace/main`. Since `//Ace/stable` is a release stream, Perforce does indeed expect change to flow to its parent by merging, so it allows the operation. But when we try merging in the reverse direction, we get an error:

```
p4 merge -S //Ace/stable -r
Stream //Ace/stable needs ‘copy’ not ‘merge’ in this direction.
```

In other words, Perforce does not expect to do any merging from `//Ace/main` to `//Ace/stable`. Copying in this direction is expected, however, so the copy command does work:

```
p4 copy -S //Ace/stable -r
```

The merge command is new in the 2011.1 release. The copy command was introduced in 2010.2, and has been retooled for streams. The `-S` flag tells these commands to operate on a stream and its parent, the default direction of flow being toward the parent. (In other words, the named stream’s parent is the target.) The `-r` flag reverses the flow.

“The merge and copy commands are special. Not only do they allow the -S flag, they check the expected flow of the stream with respect to its parent.”

Stopping the Flow

A stream's position in the hierarchy determines where change flows, and the stream type controls how change flows between it and its parent. There's also a way to control whether change flows between a stream and its parent.

I didn't show it in the previous examples, but stream specs have an Options field that fine-tunes the flow of change. Here's what Options looks like in the spec for the `//Ace/stable` stream:

```
Stream: //Ace/stable
Parent: //Ace/main
Type:   release
Options: fromparent toparent allsubmit locked
```

The "fromparent" and "toparent" options toggle whether parent-to-stream and stream-to-parent flow are expected at all. In this case, change is expected to flow in both directions—the default behavior. But if the owner of this stream changes "fromparent" to "nofromparent," for example, the copy command will refuse to copy from `//Ace/main` to `//Ace/stable`. It's not completely forbidden, however—you can always force Perforce to go against the flow. (There's a flag for that.)

Which brings me to the "allsubmit" option—the stream's owner can change that to "ownersubmit" to prevent anyone but herself from submitting changes to her stream. The "locked" option is also important; it prevents other users from modifying the stream spec. The combination gives owners control over their own streams; other, ordinary users can't thwart it. Content may be pulled from an "ownersubmit" stream into other streams, but the only way changes can get *into* the stream are through the actions of its owner.

Simple But Flexible

As you can imagine, all of this together presents some nice possibilities for guiding the flow of change in and out of streams. If it sounds complicated, fear not—almost everything you need to do with streams can be done, in most cases, using the default behaviors. For all the special and extreme cases that come up in enterprise software production, the configurable behaviors are standing by to handle them.

Now on to an equally important topic: controlling what goes into streams and workspaces.

“If it sounds complicated, fear not—almost everything you need to do with streams can be done, in most cases, using the default behaviors.”

Taking the “You” Out of Views

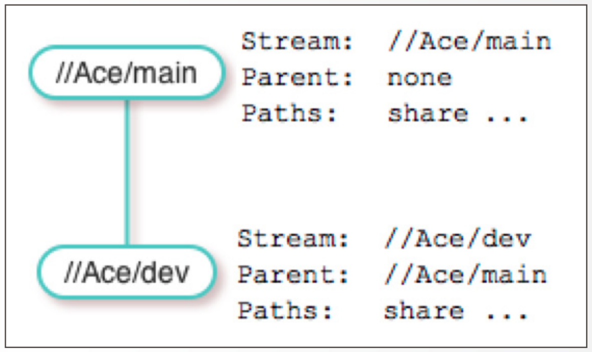
Stream views are one of the most important concepts to understand when you start working with streams. And unlike the flow of change, which is visually depicted in the Stream Graph, there’s no visual tool to demonstrate what’s happening with a stream view—yet. The stream view is a feature that’s easier to use than it is to explain, so I’m going to demonstrate with examples.

First, a little background. Perforce is designed to version large file sets. The typical Perforce repository stores far more files than you would need on disk for the work you do. Perforce “views” filter the repository files so you can see and work on only the files you need. There’s your “client view,” which maps your workspace to parts of the repo. And there are what we call “branch views,” which map one set of repo files to another for diffing and merging. If you know how views work, you can use them to derive all kinds of fantastic environments and transformations.

Streams take the “you” out of views. The Perforce Server generates workspace views and branch views for you, from stream specs. As a user, you may never see these generated views. In fact, if you’re not already a Perforce view expert, you may be wondering why I’m taking the time to write about stream views. The reason is that a lot of Perforce experts are dying to know more about them. So, for you experts out there, this chapter is for you.

A stream spec contains a template from which views are generated. The template is much simpler than the views themselves. Even better, the generated views are inherited by child streams. Creating a stream doesn't mean you have to create a template—your stream simply inherits its parent's view. And if the parent stream's view doesn't suit your needs exactly, you can edit your stream's spec and customize its view template.

Let's take a look, starting with a very simple case. Here we have two streams, `//Ace/main` and `//Ace/dev`, along with their specs:



```
Stream:   //Ace/main
Parent:   none
Paths:    share ...

Stream:   //Ace/dev
Parent:   //Ace/main
Paths:    share ...
```

Figure 8: Stream path

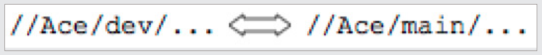
What you see in both specs is the default view template—the entire stream path is "shared." (I'll explain this in a bit.) How does this affect client views? Well, when you switch your workspace to the `//Ace/main` stream, its client view becomes this:

```
//Ace/main/...  ⟺  //your_ws/...
```

The client view maps the root of `//Ace/main` to the root of your workspace. Same thing happens after switching your workspace to `//Ace/dev`—now its view is this:

```
//Ace/dev/...  ⟺  //your_ws/...
```

As I mentioned in a previous section, you can use commands like `p4 copy -S //Ace/dev` to integrate between a stream and its parent. The `-S flag` says "use this stream's branch view, " the view that maps the stream to its parent. `//Ace/dev's` branch view looks like this:



```
//Ace/dev/... ⟺ //Ace/main/...
```

This is due to the default view template, which says the entire stream path ("...") is to be shared with client views and branch views. Because of that, the entire stream can be synced to workspaces, and the entire stream can be branched and integrated.

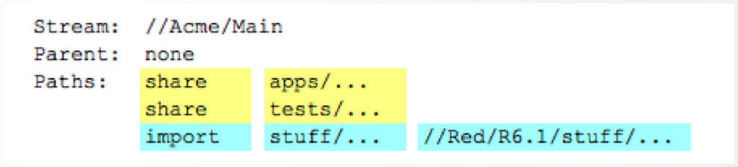
This is simple, easy to use, and perfectly suitable for small projects. For large projects, it isn't practical, of course. So, what are the alternatives?

Streams offer a choice of behaviors for each path (and by "path" I mean any level of subdirectory):

- **Share** changes with other streams. These paths appear in both client views and branch views, and are mapped to the stream. The effect, as I just mentioned, is that files can be synced, submitted, and integrated.
- **Isolate** changes from other streams. These paths appear only in client views and are mapped to the stream. That means you can sync and submit the files in these paths, but you can't integrate them. This is useful for built binaries, generated docs, and other content that you want checked in but not branched or merged.
- **Import** a path from elsewhere. These paths appear only in client views and are mapped per the parent stream's view (the default), or to another depot location. You can sync these files, but because they're not mapped to the stream itself, you can't submit them. Nor can you integrate them. All they do is mirror other paths.
- **Exclude** a path from both client views and branch views. This keeps unneeded content out of workspaces and streams entirely.

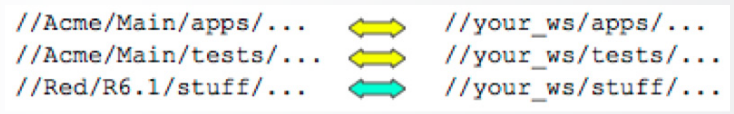
Here's an example from Massively Large Media Corporation (MLMC), a fictional company. MLMC is a conglomerate of three separate software companies: Acme Systems, Red Resources, and Tango Tools. Each company makes a sizable product, and all three contribute to MLMC's bundled products. The files for these components are housed in three separate depots, `//Acme`, `//Red`, and `//Tango`.

The Acme mainline is configured thus:



```
Stream:  //Acme/Main
Parent:  none
Paths:   share     apps/...
         share     tests/...
         import    stuff/...     //Red/R6.1/stuff/...
```

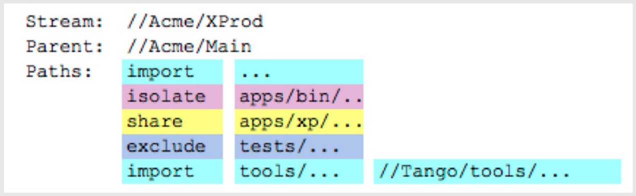
I've color-coded the view template so you can recognize the effect each path has on the generated views. Here, for example, is the client view you'd get if you were to switch your workspace to the `//Acme/main` stream:



```
//Acme/Main/apps/...    ⟺   //your_ws/apps/...
//Acme/Main/tests/...   ⟺   //your_ws/tests/...
//Red/R6.1/stuff/...    ⟺   //your_ws/stuff/...
```

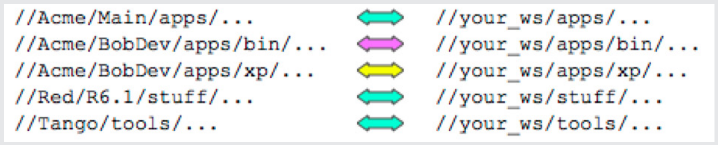
It's pretty easy to see what's going on here. The stream spec's Paths field lists folders relative to the root of the stream. Those are the folders you'll get in your workspace, beneath your workspace root. The shared folders are mapped to the `//Acme/Main` path, and the imported path is mapped to its location in the `//Red depot`.

As it turns out, no one does actual development in the Acme mainline. Take the XProd feature team. They have a development stream of their own, defined thus:



```
Stream:  //Acme/XProd
Parent:  //Acme/Main
Paths:   import     ...
         isolate    apps/bin/..
         share      apps/xp/...
         exclude    tests/...
         import     tools/...     //Tango/tools/...
```

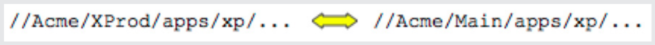
Switching your workspace to the `//Acme/XProd` stream gives you this view:



```
//Acme/Main/apps/...          ⟷   //your_ws/apps/...
//Acme/BobDev/apps/bin/...    ⟷   //your_ws/apps/bin/...
//Acme/BobDev/apps/xp/...     ⟷   //your_ws/apps/xp/...
//Red/R6.1/stuff/...          ⟷   //your_ws/stuff/...
//Tango/tools/...             ⟷   //your_ws/tools/...
```

Here we see stream view inheritance at work. Imported paths are mapped to whatever they're mapped to in the parent's client view, unless they are given other paths to import, as in the case of tools. The shared and isolated paths are mapped to the child stream; these contain the files the XProd team are working on and will be submitting changes to. And the excluded path doesn't appear in the workspace view at all.

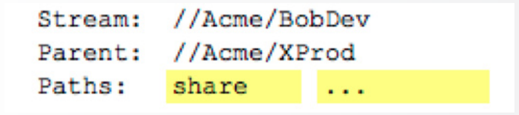
Because the `//Acme/XProd` stream has a parent, it has a branch view that can be used by the copy and merge commands. That branch view is:



```
//Acme/XProd/apps/xp/...  ⟷  //Acme/Main/apps/xp/...
```

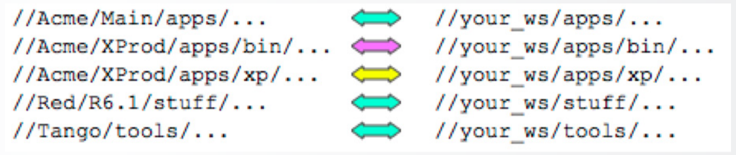
Wow! All those paths in the `//Acme/XProd` view, and only apps/xp can be branched? That's right—it's the only path that is shared in both streams. When you work in an `//Acme/XProd` workspace, it feels as if you're working in a full branch of `//Acme/Main`, but the actual branch is quite petite.

Now, let’s go one level deeper. Suppose Bob, for example, creates a child stream from `//Acme/XProd`. His stream spec looks like this:



```
Stream:  //Acme/BobDev
Parent:  //Acme/XProd
Paths:   share    ...
```

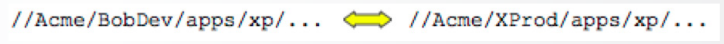
Note that Bob’s stream has the default view template. Given that Bob’s entire stream path is set to “share,” will his entire workspace be mapped to his stream? No, because inherited behaviors always take precedence; sharing applies only to paths that are shared in the parent as well. A workspace for Bob’s stream, with its default view template, will have this client view:



```
//Acme/Main/apps/...          ⟺   //your_ws/apps/...
//Acme/XProd/apps/bin/...     ⟺   //your_ws/apps/bin/...
//Acme/XProd/apps/xp/...      ⟺   //your_ws/apps/xp/...
//Red/R6.1/stuff/...          ⟺   //your_ws/stuff/...
//Tango/tools/...             ⟺   //your_ws/tools/...
```

As you can see, a workspace in Bob’s stream is the same as a workspace in the XProd stream, with one exception: The paths available for submit are rooted in `//Acme/BobDev`. This makes sense; if you work in Bob’s stream, you expect to submit changes to his stream.

By contrast, the branch view that maps the `//XProd/BobDev` stream to its parent maps only the path that is designated as shared in both streams:



```
//Acme/BobDev/apps/xp/...   ⟺   //Acme/XProd/apps/xp/...
```

The default template allows Bob to branch his own versions of the paths his team is working on, and have a workspace with the identical view of nonbranched files that he’d have in the parent stream. And Bob didn’t have to lift a finger to create it!

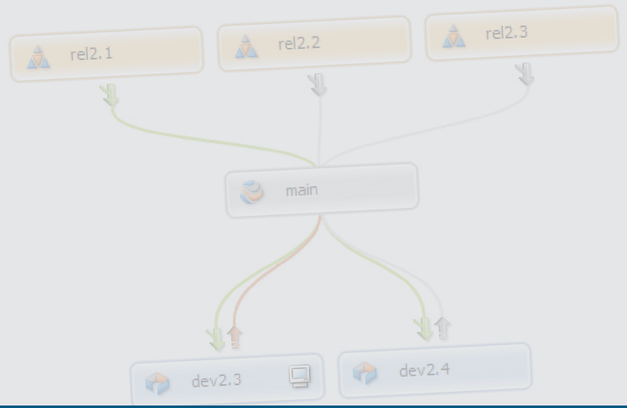
I hope these simple examples were clear, yet varied enough to hint at the kinds of things you can do with stream views. One key point is that child streams can’t branch more than their parent streams are willing to share. Another is that stream views default to the parent view. This gives control to owners of parent streams without completely tying the hands of developers who branch their own child streams.

The Paths field is only one part of the stream view template, by the way. The template also offers a way to define remapped and ignored files, the effects of which are inherited by child streams. You'll be happy to know, for example, that a mainline can be configured to forbid anyone, in any related stream, from checking in files with names like "`.*.swp`"!

Conclusion

That wraps up our short introduction to Perforce Streams. Streams are branches with brains: related sets of files with some understanding of the relative stability, which changes to merge, where those changes go, and how they get there. The flow of change is modeled by stream parent-child relationships, types, and flow control options. The stream composition is modeled by the stream view and paths.

We're very excited about the power, simplicity, and flexibility of streams. Streams do not define a rigid process. Rather, they try to elegantly model a sensible way of working with branches, even in complicated situations. That should make branching, merging, and other version control tasks easier. And by making the power of Perforce version control more accessible, they can make your development process more responsive and agile.



FAQ

Here are a few common questions and answers about Streams.

When will Streams be available?

Streams is available now for download. Learn more about the latest features in Perforce 2011.1 with Streams.

What will performance be like?

“At least as good as p4 integ,” is what I used to say when I worked with the design team on this project. But our engineers have really overshot the mark. Their focus on performance has made Streams more efficient than classic codelines for most uses.

Here’s an example: The Perforce Server has always been smart about storing a history of integration data. (“Integration” being branching, copying, and merging.) Now, using streams as a frame of reference, the server caches pending integration data as well. When you query to find out what needs to be integrated between a stream and its parent, the server response time varies from “pretty darn quick” to “right away.”

Will Streams support private branching?

Another product Perforce is working on, completely outside of Streams, is private local branching. The public beta of P4Sandbox is scheduled for release at the end of Q1 2012.

Meanwhile, as a little eating-our-own-dogfood project, we’ve set up a Perforce server with triggers that give owners complete control over access to their own streams. In addition to flushing out a covey of prerelease bugs, this project is proving that streams are a great framework for access control. If what you want is private branching in the server, streams, plus triggers and protections, are a nice solution.

Can you do branching in place with Streams?

Yes. "Branching in place" is the ability to branch files and switch your workspace between branches without actually touching your local files. It's an extremely efficient operation when switching between branches whose files are largely identical.

A very satisfying byproduct of the engineering that went into Streams and local private branching is that p4 client (alias p4 workspace) and "p4 update" (alias p4 sync –s) now have behaviors that support branching in place. These behaviors are available with or without Streams.

Are streams lightweight branches?

A stream can be a very lightweight branch if you use its view to limit the branchable components of the parent stream; the rest of the parent stream is simply mirrored in the child stream's workspaces.

Otherwise, branching stream files is just like branching classic Perforce files. The server doesn't actually copy the files themselves. Instead, it makes "lazy copies"—database pointers to branched files. This is not as lightweight as in systems where a branch is a single database pointer, and it can be a problem if you're branching huge file trees very frequently. Stream views are designed to nip this problem in the bud.

That being said, our server developers have been tossing around an idea for using the stream as a container for bona fide lightweight branching. This is but a gleam in their eyes at the moment; we'll see what happens with it.

How do streams enforce good behavior?

Streams give users a roadmap of where and how to do branching and merging. They guide good behavior rather than enforce it. Some commands, like p4 copy and the new **p4 merge**, can enforce the intended flow of change between streams. The classic **p4 integ** command can be used to "go against the flow," however. The only thing a nonprivileged stream owner can do about that is to configure the stream to reject other users' changes. For industrial-strength enforcement, triggers and protections will do the job.

For more information go to perforce.com