

# From the Files of a Perforce Consultant...

**Jeff A. Bowles**  
**Piccolo Engineering, Inc.**  
**jab@piccoloeng.com**

*When a Perforce customer calls a Perforce consultant, each situation is unique. This paper will try to call attention to some of things common to many customers – questions they've asked, situations they are anticipating but hope to avoid, etc.*

---

As a Perforce consultant, I see many approaches to common situations. When a potential client calls or e-mails, they

The initial questions tend to fall into these sorts of categories:

- “Could you help us use Perforce effectively?”
- “We need a tune-up.” – e.g. dealing with branches, testing backup procedures;
- “We’re doing planning for a future release, development project, etc.”

(One of the frequent requests is “could you install Perforce for me?” It’s easy to understand, but [as you know] the product is easy enough to install that I usually just point them at certain URLs and tell them to save the consulting dollars for the more advanced topics in a couple of months – or training for their users.

## **Where To Start?**

When arriving at a customer’s, I often complete a form<sup>1</sup> that contains questions like these:

- “Who is the administrator?” (I.e. “Who reboots the server when it crashes?”)
- “What’s your backup strategy?” (I listen to make sure they mention the word “journal” or “transaction log.”)
- “Tell me how you will issue a release using Perforce.” (The follow-up question is, roughly, “How will you issue a patch against that release?”)

The answers to these questions leads to the real needs, e.g. “training” or “re-thinking branching strategies” or “getting IT buy-in for Perforce administration.”

If there’s not a set of answers to these sorts of questions, either the list of questions needs adjustment or the first order of business is getting everyone to agree on these basics.

---

<sup>1</sup> A version is included at the end of this paper, sometimes useful for a new installation.

### An example – of a conversation you don't want to hear

Let's say that the Perforce administrator's name was Dweezle, and the IT head was Moon Unit. It can be telling when you ask Dweezle about the backup strategy, and he responds "Moon worries about that part", and when you ask Moon Unit, her response is "That's part of Dweezle's position."

### Using Perforce Effectively – where to put the new project?

Of course, there's always the questions that anticipate a new need. Checking in new code (or a new project) is the main one, so let's start there.

You can add them anywhere – the next section talks more about directory structures and the like. Once you've picked the pathname...

1. "p4 add" is good enough...
2. but put the files into the right directory structure first: there is no reason to add first, then immediately rename, except perhaps to preserve a history.
3. "ant" or other incrementally compiled Java environments can help identify source files that live in the wrong place in the tree.

### Review of "mainline" strategy

Where you put files can make future work much easier or harder. Often, a customer will ask what models work well with Perforce, and it's useful to review the one that Perforce customers use frequently.

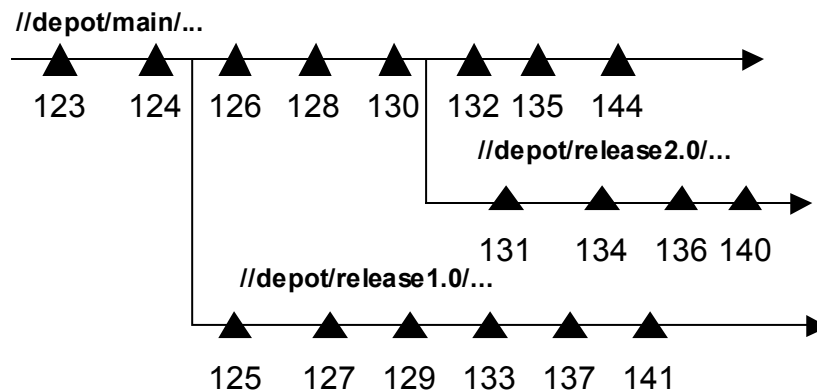
The Perforce white paper, *High-level Best Practices in Software Code Management*<sup>2</sup>, presents a way to layout codelines (also called "branches") that isn't specific to Perforce, but that can leverage many of Perforce's strengths. A simple version of this strategy is to have one codeline on which you do your work:



As you issue releases, you can then create child codelines to house them:

---

<sup>2</sup> This white paper, which can be found <http://www.perforce.com/perforce/bestpractices.html>, was presented by Laura Wingerd and Christopher Seiwald at *Eight International Workshop on Software Configuration Management in Brussels, July 1998*.



These diagrams can become somewhat complex – it’s possible to have one branch per release (the above shows a ‘main’ and two releases) – but the basic model is always the same:

1. A codeline/branch has one purpose at a given moment. It might be “the codeline for development of feature X”, “a codeline for issuing release 1.0.2 patches,” or “the main conduit, called ‘mainline’, for integrating work and bug fixes from one codeline to another.” **If a codeline/branch starts serving two needs, it’s time to make a new branch or to change how people are using that codeline/branch.**
2. The person running “p4 resolve” (doing integrations) tries to stick to the model of “push from parent to child, or child to parent”. This keeps data flow simple, since you never encounter the ‘patched in 1.0, fixed in 2.0, accidentally not included in 3.0’ embarrassment.
3. The lazy-copy mechanism in Perforce, called Inter-file Branching, has very low overhead on the server. The data storage for a new codeline created using “p4 integrate” is entirely in the relational database files, because the new codeline/branch is a virtual copy.

**It’s helpful to have a chart like the above, for whatever structure is in use, for reference.**

If you have no strong opinions about which model to use, this ‘mainline’ model is excellent because it’s already documented, is a model that is well understood and easily described to others, and is easy to hand off to another Perforce-savvy administrator. (If you have another model in mind, make sure that your model addresses the technical issues that *High-level Best Practices* solves.)

### **The Tune-Up Request: Renaming files**

Renaming files in Perforce is straightforward: you integrate to the new location, delete the old one. Renaming a file in a parent codeline presents a problem: should copies in children branches map to the old [and obsolete] name or the new one, in the parent? (A variation of this is “do you want to propagate the rename into a child that’s already issuing releases and patches?”)

One of the exercises in the Perforce classes presents a strategy for this: annotate the *named branch specification* to do this. A basic *branch specification* might be:

View: //depot/main/... //depot/release1/...
<b>Original branch specification for 'release 1 codeline'</b>

If, for example, you'd renamed a file from "barney.cpp" to "fred.cpp" in your mainline, you'd end up with the following:

View: //depot/main/... //depot/release1/... //depot/main/src/fred.cpp //depot/release1/src/barney.cpp
<b>Modified branch specification for 'release 1 codeline'</b>

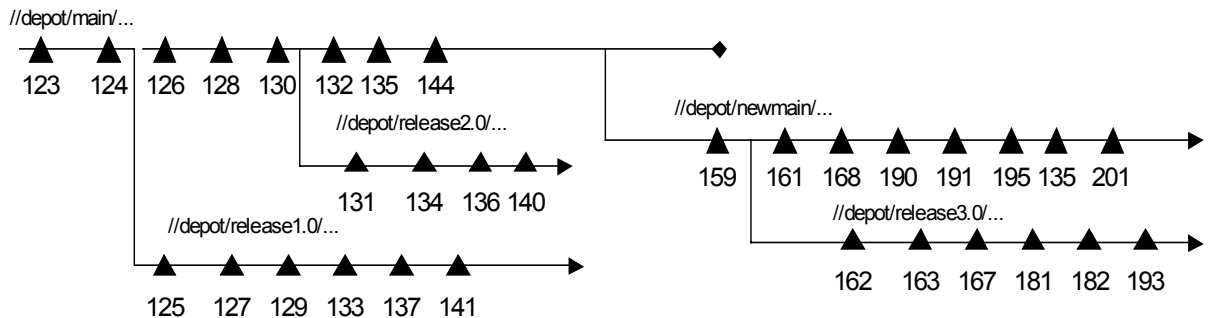
Note that Perforce doesn't do this automatically: you need to run "p4 branch" to add this second line. It won't automatically make main/.../fred.cpp the parent of its release1 counterpart, but the next time you run "p4 integrate -b br-spec" for that codeline, main/src/fred.cpp will be related to release1/src/barney.cpp.

### The Tune-Up Request: reorganizing the 'mainline'

Sometimes it's easiest to make a "//depot/newmain/..." (branched from the older "//depot/main/...") that is in the new structure. For example, someone might've decided to have a branch strategy that has simply failed to work for your needs, and you want to have a new strategy for future work<sup>3</sup>.

The simplest case is renaming individual files – the previous section addresses that. (Renaming a couple of directories is the same solution.)

For major work, it's some-times useful to make a new "main codeline" called "//depot/newmain/..." to have a clean place to start.



<sup>3</sup> "1.0 is parent of 2.0 is parent of 2.5 is parent of..." has caused a number of headaches, and is a terrible strategy for many reasons. The easiest reason is that it's simple to explain to middle management, but leaves no room for emergency releases and no room for propogating the 1.0 bug fixes into 4.0 without polluting every release between the two. (You want the option of including, or not including, those 1.0 bug fixes into 2.0 and 2.5; this strategy takes that choice away.)

This way, ‘main’ remains the parent of older releases (1.0 and 2.0) – bug fixes from those releases can be propagated to ‘main’ first, then to ‘newmain’ and eventually into the 3.0 (or later) codelines<sup>4</sup>.

## Component-ware

Sometimes, there are separate groups working on the components for a product – the Core Library Group, the GUI Group, and so on. Each group “releases” the product 1 and you will build your überproduct using these subcomponents.

For example, App 4.0 consists of:

- CoreLib version 1.0
- GUILib version 1.3
- GraphicsLib version 2.56
- AppSrc version 4.0

For this example, let’s say they’re identifying the source they want you to use, with labels.

Commands to create this “App 4.0” codeline might be:

```
p4 integ //depot/main/core/...@core-v1.0 //depot/app4.0/core/...
p4 integ //depot/main/gui/...@gui_v1.3 //depot/app4.0/gui/...
(and so on)
p4 submit
```

Of course, you’d use a branch specification to do this (“p4 branch”), for integrations both directions<sup>5</sup>.

This has several advantages: it tracks history and creates a consistent source tree for app 4.0. The integration engineer needs to know which versions of which subcomponents are in “App 4.0” but the developer usually won’t have to worry about it.

## How Do I Restore From a Catastrophe?

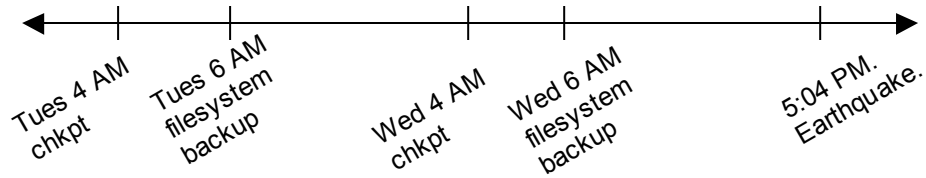
Most administrators understand the idea that you do backups anticipating the worst. In California, the “worst case” is the building collapsing as the state falls into the Pacific<sup>6</sup>.

---

<sup>4</sup> If you find that you want to do major restructuring of files/directories after every release, then there’s another problem to solve first – you’ll spend more time dealing with the overhead of these processes than with development itself! Best to ask “why are we doing this, again?” and go from there.

<sup>5</sup> The p4 integrate command won’t be that bad once the branch specification’s made. For example, the first integration in the list – the one that pulls main/core/...@core-v1.0 content into //depot/app4.0/core/... – would be “p4 integ -b br-spec //...@core-v1.0”. This assumes that no files except those from //depot/main/core/... are in the label.

<sup>6</sup> In Chicago, it’s fear of the celebratory riots when the Cubs finally win a World Series. The San Francisco scenario, we agree, is more likely.



(Any backup question requires an immediate followup: do you need items recovered right now, or is this for planning purposes?)

In the above diagram, how fast can we recover – and to what moment can we recover?

If the backups are on-line (at a remote fileserver location) you could retrieve the backups within minutes. **If your IT department uses a standard filesystem backup mechanism, it could be minutes and could be weeks.** Let's assume that you have a backup (of checkpoint/journal/depot trees) available. Then...

1. You can recover to 6 AM data easily, using checkpoint and journal and depot/\* trees copied at 6 AM.
2. You can recover to 4 AM's checkpoint just as easily, just from the checkpoint and depot/\* trees. Although this isn't as practical (you have the extra two hours, why not include them in the metadata restore) it shows that you can use the 4 AM checkpoint and the 6 AM depot/\* trees to make a working server that reflects 4 AM data. **The depot/\* tree is allowed to be later than the metadata you restored, but not vice versa.**

But you've still lost 11 hours of development work. (Admittedly, if the building's gone, this might be the least of your worries. Unless you're doing financial work for a bank or Wall Street, this won't bankrupt the company.)

A variation of this might be to incrementally copy the journal and depot/\* tree every few minutes during the day, to a trusted off-site server. If you did this every 30 minutes, then you could restore to the 5:00 PM or 4:30 PM copy of the data.

There are a few things that need to be included as part of disaster planning:

1. Journaling should be turned on for every production environment. The Windows installer enables this for Windows (but check); it has to be done explicitly on Unix servers.
2. Are the restore procedures written down? Is there a hard-copy available to the folks who will do that restore?
3. Disks go bad, too. If your checkpoint, journal, and db.\* files are on the same physical drive and it explodes, life will be too interesting for comfort. Split them up.

### How do I store a web site?

This is frequently asked. There are FAQs on this, that suggest strategies like these:

1. If you're using Apache, map *DocumentRoot* area into Perforce as a workspace, then run "p4 sync" frequently (or when files updated in Perforce) to update the on-disk area that Apache serves to the web users.
2. Use Apache plug-in (WebKeeper) on Unix. This doesn't require an on-disk area for the Perforce files, since it automatically pulls them from Perforce when they're requested.
3. Use "p4ftpd" to update files in Perforce from your HTML editor, use one of the other [two] strategies to then get those changes to the web users.

### **Other items in the files: Cygnus.**

"Cygnus" is a collection of freeware command-line tools that look like Unix commands on Windows/NT. (The MKS Toolkit is a commercial counterpart that's more robust.)

The *Cygnus Toolkit* isn't without its problems – there are problems in particular builds and releases, and its filename syntax is something like "/hdd/c/bin/file.exe". **While there is a "p4.exe" that's linked to the Cygnus libraries, there is not [at this time] a "p4win.exe" counterpart** - so it's not easy to switch transparently between Perforce command-line and Perforce GUI in a Windows+Cygnus environment.

*Workaround #1: Use MKS instead. It isn't free, but avoids these problems.*

*Workaround #2: Beginning with release 2001.1, sharing workspaces between Unix and NT is easier. If you need Unix functionality for convenience, you might consider having the workspace live on a Unix machine and using the Perforce Windows GUI as necessary.*

### **Other items in the files: sharing workspaces.**

See 2001.1 release notes. The approach that is simplest is to use "share" in workspace options as necessary.

### **Other items in the files: multiple sites involved in development.**

Here is an excerpt from a comparison of Perforce *remote depots* and ClearCase "multisite":

1. A ClearCase Multisite installation gives you, the local user, a snapshot of a remote site's work in a readonly, local branch,
2. Including labels, including metadata.
3. A Perforce remote depot gives you live readonly access, real-time, of a remote server's data.
4. But not labels, usernames, metadata.

**Neither is a seamless development environment: both ClearCase's product and Perforce "remote depots" assume that development on a given file happens at exactly one site.**

It's important to know which configuration of the product works for which situation:

- If people from Paris and San José are both developing `src/lib/x.c`, then they should expect to connect to the same server.
- If people from Paris are developing `src/lib/x.c` and folks from San José are using it to build another library or application, then remotely pulling the file from Paris to San José using *remote depots* makes sense.

The same two points could be made about ClearCase's Multisite product.

The main difference, in practice, between ClearCase's remote product and Perforce remote depots is when the remote data is retrieved:

- With Perforce, it's retrieved as it's requested. If you ask for `//paris/src/lib/x.c#5` it's retrieved when you ask for it. (This is good in that it's live data taken that instant from Paris; bad in that it forces you to wait for data packets coming from around the planet.)
- With ClearCase, it's cached into a *readonly branch* that's imported explicitly by an administrator.

In Perforce, the San José administrator could imitate this (for his users) by importing the data into a branch, e.g.

```
p4 integ //paris/src/lib/... //depot/copy-of-paris/src/lib/...
p4 resolve -at //depot/copy-of-paris/src/lib/...
p4 submit
```

He'd probably mark this "readonly" (using 'p4 protect') to avoid tempting people to check changes into `//depot/copy-of-paris`.

Creating a *readonly branch* that imports the remote data, perhaps at 3 AM every morning, is a straight-forward mechanism to share results of software development without dealing with slow network lines between servers.

## Conclusions

The basic questions that come up for the Perforce Administrator, as he/she is setting up a site or writing scripts/procedures to make it an "easy" job, tend to be questions others have already asked. Your answers may be different, because your needs vary. It's always a good idea to know what questions the other folks asked.

## **Appendix – Example of a “Server Set-up Reminders” list**

1. What’s the codeline topology to be?
  - (a) Is there a mainline model?
  - (b) Is there a sandbox for each user?
  - (c) Is there an old-releases/... area?
  - (d) (Evil) Do you want to implement the “no child/child integrations” trigger?
  - (e) Have you created brach specifications for all codelines that exist?
2. What client specifications exist?
  - (a) Are there any default client specifications to make?
  - (b) Build client specifications?
3. Run p4 protect and set up default security protections.
4. Set up some sort of script for doing regular backups and checkpoints.
  - (a) Are the db.\* files and the checkpoints and journals on the same disk?  
(They should not be.)
  - (b) Is the depot/\* tree on the same disk are the journal? (Probably should be.)
  - (c) Is anything incrementally copied [to another machine] during the day?
5. What’s the name/IP of the backup machine? Is there a Perforce license for that machine yet?
6. Where’s the intranet server? Put the FAQs there.
7. Have you downloaded and set up the mail review mechanism?
8. Set up a ”build client” and work on the makefiles for that. Are you creating overnight builds from labels or changelists?