arch Meets hello-world

A Tutorial Introduction to The arch Revision Control System

visit the Hackerlab at www.regexps.com

---

by
Thomas Lord

Contents

Introducing arch

up: arch Meets hello-world
next: System Requirements

`arch` is a revision control, source code management, and configuration management tool.

This manual is an `arch` tutorial: its purpose is to help you get started using `arch` for the first time, and then learn some of the more advanced features of arch.

Who is this Manual For?

In order to use this manual, you should be familiar with the basic unix command line tools (such as `ls` , `mv` , and `find`).

In addition, you should be familiar with the programs `diff` and `patch` and the concept of a `patchset` .

It is very helpful, but not strictly necessary if you have used or are at least familiar with other revision control systems such as `CVS` .

Where's the Reference Manual?

`arch` is largely a *self documenting* program. The command:

```
        % tla help
```

will provide you with a categorized list of all available commands, and for a given command `foo` ,

```
        % tla foo -H
```

will provide you with documentation for that command.

Another Source of Help -- the Mailing List

Arch is sufficiently different from older and competing systems that new users are often a bit disoriented for the first few days. You may find it helpful to seek help on the `gnu-arch-users` mailing list which you can find via links from:

```
http://www.gnu.org/software/gnu-arch
```

What is Revision Control?

A "revision control system" is a librarian and coordination tool for trees of files and the changes made to them. For example, a typical software project uses revision control to keep track of how the project's source code evolves over time, to keep track of each change to that code (such as each bug fix or feature addition), to share those changes among all the programmers working on the project and help them remain in sync, and to combine changes made at different times and/or by different programmers into a single source tree.

A "source management tool" is one that helps you to manage large source trees even if they have many more files that you can keep track of "by hand". For example, a source management tool can *inventory* the source files in a tree, distinguish the source files from scratch files and and other files that maybe stored there, and inform you when source files are added and deleted.

"Configuration Management" addresses the needs of projects which combine multiple, separately maintained source trees into a single tree. A configuration management tool helps you to easily construct the combined project and to keep track of how development on the component parts is synchronized.

Why Use arch?

`arch` has a number of advantages compared to competing systems. Among these are:

Works on Whole Trees `arch` keeps track of whole trees -- not just individual files. For example, if you change many files in a tree, `arch` can record all of those changes as a group rather than file-by-file; if you rename files or reorganize a tree, `arch` can record those tree arrangements along with your changes to file contents.

Changeset Oriented `arch` doesn't simply "snapshot" your project trees. Instead, `arch` associates each revision with a particular *changeset*: a description of exactly what has changed. `arch` provides changeset oriented commands to help you review changesets, merge trees by applying changesets, examine the history of a tree by asking what changesets have been applied to it, and so forth.

Fully Distributed `arch` doesn't rely on a *central repository* . For example, there is no need to give write access to a project's archive to all significant contributors, instead, each contributor can have their own archive for their work. `arch` seamlessly operates across archive boundaries.

System Requirements

up: arch Meets hello-world
next: arch Commands in General
prev: Introducing arch

In order to use `arch` , there are some software tools that you must already have available.

Tools Used to Build arch

GNU Make You will need `GNU Make` in order to build `arch` .

Standard Posix Shell Tools The package framework (i.e., the configure and build process) assumes that some standard Posix shell tools are available on your system:

```
        awk     find    mkdir   sh      wc
        cat     fold    printf  tee     xargs
        chmod   grep    pwd     test
        date    head    rm      touch
        echo    ls      sed     tsort
```

Note: On some systems, the program installed as `/bin/sh` is *not* a Posix shell (it may be a variant of `csh` or a very buggy implementation of Posix `sh`). On such systems, you should use a different shell to run `configure` , such as:

```
  % /usr/local/bin/bash ../configure --config-shell /usr/local/bin/
bash
```

The null Device Your system must have `/dev/null` . Information directed to `/dev/null` should simply disappear from the universe. As a special "Green Software" measure, we have made provisions that will enable your computers to convert that discarded information into *heat*, which you may use to supplement conventional heating systems.

Tools Used Internally by arch

The remaining tools are used internally by arch itself. They don't necessarily need to be on your `PATH` -- when you build `arch` from source, run the configure script:

```
% ./configure --help
```

and

```
% ./configure --help-options
```

for information about how to point `arch` to the correct versions.

GNU Tar You must have `GNU tar` . `arch` invokes `tar` internally to pack and unpack files that it stores in archives. It is important that all versions of `arch` use a compatible version of `tar` , for which purpose `GNU tar` was chosen.

GNU diff and GNU patch After much deliberation, I've decided to go ahead and rely on the **GNU** versions of `diff` and `patch` . Specifically, you need a version of `diff` that can generate "unified format" output (option `-u`) and a version of `patch` that understands that format and that understands `--posix` . (It would be trivial to use "context diffs" and, thus, standard `diff` and `patch` , however, unified diffs are much easier to read, and I'm hoping that picking specific implementations of these critical sub-components will help contribute to the long-term stability of `arch` .)

arch Commands in General

up: arch Meets hello-world
next: Introducing Yourself to arch
prev: System Requirements

Every command in `arch` is accessed via the program `tla` , using an ordinary sub-command syntax:

```
% tla <sub-command> <options> <parameters>
```

A list of sub-commands can be obtained from:

```
% tla help
```

A brief summary of the options to any command is given by:

```
% tla <sub-command> -h
```

A more detailed help message for each command is given by:

```
% tla <sub-command> -H
```

For example, try:

```
% tla my-id -H
print or change your id
usage: tla my-id [options] [id]

  -h, --help     Display a help message and exit.
  -H             Display a verbose help message and exit.
  -V, --version  Display a release identifier string
                 and exit.
  -e, --errname  specify program name for errors
  -u, --uid      print only the UID portion of the ID
```

```
        With no argument print your arch id.

        With an argument, record ID-STRING as your id
        in ~/.arch-params/=id

        Your id is recorded in various archives and log messages
        as you use arch.  It must consist entirely of printable
        characters and fit on one line.  By convention, it should
        have the form of an email address, as in this example:

                Jane Hacker <jane.hacker@gnu.org>

        The portion of an id string between < and > is called your
        uid.  arch sometimes uses your uid as a fragment when
generating
        unique file names.

        The option -u (--uid) causes only the uid part of your id
string
        to be printed.
```

There is a great deal of regularity among commands regarding option names and parameter syntax. Hopefully, you'll pick this up as you learn the various commands.

Introducing Yourself to arch

up: arch Meets hello-world
next: Creating a New Archive
prev: arch Commands in General

The first step to using arch is to set your id with a command like:

```
        % tla my-id "Tom Lord <lord@emf.net>"
```

Your id should be your name, followed by your email address in angle brackets.

`arch` records your id in various log messages that it creates.

You can find out your id with:

```
        % tla my-id
        Tom Lord <lord@emf.net>
```

How it Works -- Your arch Id

After the command above, you will have some new files in your home directory:

```
        % ls ~/.arch-params
        =id

        % cat ~/.arch-params/=id
        Tom Lord <lord@emf.net>
```

Caution: You usually should not edit files in `~/.arch-params/` "by hand."

Creating a New Archive

up: arch Meets hello-world
next: Starting a New Project
prev: Introducing Yourself to arch

An archive is a dedicated directory which `arch` uses to hold a library of your project trees and changesets. This chapter shows you how to create a new archive.

Choose a Location

You need to decide where to store your archive: where to create the directory that will contain the archive.

Usage Advice: It is likely that you'll eventually want to have *more than one* archive. Therefore, it is a good idea to create a directory of archives.

In the examples that follow, we'll be creating an archive as a subdirectory of `~/{archives}` , a directory of archives.

```
        # Create a directory in which to store archives:
        #
        % mkdir ~/{archives}
```

Choose an Archive Name

Next, you need to choose a name for your archive. An archive name consists of an email address, followed by two dashes (`--`), followed by a suffix. By convention, the email address should be that of the archive owner.

In the example, we'll use the name:

```
        lord@emf.net--2003-example
```

Usage Advice: If you use a single archive for a very long time it will eventually accumulate a very large amount of data and thus start to become inconvenient to work with. Because `arch` seamlessly operates across archive boundaries, there is no need to keep everything in just one archive. It's a good idea to plan to divide up your archives by *time* and that suggests that you include a date in the archive name. In the example above, the archive is labeled `2003` : a year later, we could create `lord@emf.net--2004-example` and continue the project in that new archive. The `2003` archive will still exist at that point -- we'll just stop adding new data to it.

Usage Advice: You should plan on having multiple archives, and therefore choose archive names that distinguish them. The suffix `-example` above tells us that this archive is being created just work through the examples in this tutorial.

Create the Archive

To create a new archive, use the `make-archive` command, telling it the archive name and archive location:

```
# Create the new archive
#
% tla make-archive lord@emf.net--2003-example \
                     ~/{archives}/2003-example
```

Make this Your Default Archive

To save yourself from having to type the archive name to every future command, declare that your new archive is your default choice:

```
# Choose a default archive
#
% tla my-default-archive lord@emf.net--2003-example
```

Your current default is reported by:

```
% tla my-default-archive
lord@emf.net--2003-example
```

And you can cancel the default setting with:

```
% tla my-default-archive -d
user default archive removed
```

(If you experiment with `-d` , be sure to re-establish your default archive so that you can continue to follow the examples.)

How it Works -- New Archives

Let's examine what that command did.

First, `tla` now knows about the new archive:

```
# What archives does `tla' know about?
#
% tla archives
lord@emf.net--2003-example
        /home/lord/{archives}/2003-example


% tla whereis-archive lord@emf.net--2003-example
/home/lord/{archives}/2003-example


# Where is that data stored?
#
% ls ~/.arch-params
=default-archive        =id             =locations

% cat ~/.arch-params/=default-archive
lord@emf.net--2003-example

% ls ~/.arch-params/=locations
lord@emf.net--2003-example

% cat ~/.arch-params/=locations/lord@emf.net--2003-example
/home/lord/{archives}/2003-example
```

Next, the archive directory has been created and contains a few files:

```
% ls ~/{archives}
2003-example
```

```
% ls -a ~/{archives}/2003-example
.                         .archive-version
..                        =meta-info

% cat ~/{archives}/2003-example/.archive-version
Hackerlab arch archive directory, format version 2.

% ls -a ~/{archives}/2003-example/=meta-info/
.        ..       name

% cat ~/{archives}/2003-example/=meta-info/name
lord@emf.net--2003-example
```

Caution: You usually should not edit files in `~/.arch-params/` or files in an archive "by hand."

Starting a New Project

up: arch Meets hello-world
next: Starting a New Source Tree
prev: Creating a New Archive

This and later chapters will show you how to set up and manage a simple project with `arch` through the specific example of a *hello world* program.

Choose a Project Category

As a first step, you must choose a general category to serve as a name for the project. In the examples, we'll use the name:

```
        hello-world
```

Choose a Project Branch

`arch` encourages you to divide up the work on a project into separate *branches* .

Roughly speaking, branches are a mechanism for splitting the work on a project into two or more, largely independent efforts. Let's suppose, for example, that the `hello-world` project has two needs:

1) A need to make regular releases of good ol' fashioned `hello-world` , fixing simple bugs, porting the program, and adding tiny features.

2) A need to begin work on a graphical user interface for `hello-world,` which is expected to take about a year to complete.

We'd like those two efforts to proceed in parallel, but not get in each other's way. For example, we don't want GUI code to appear in the regular releases until it is working fairly well.

In such a case, we'll use branches: one for regular releases (the *mainline* branch) and another for GUI features (the *gui* branch).

There are many other uses for branches, some of which will be described later in the manual. For now, we just need one branch: a branch for the official latest sources of `hello-world` , which we'll call:

```
        hello-world--mainline
        ^^^^^^^^^^^  ^^^^^^^^
           |            |
           |       branch name
      category name
```

Notice that the category and branch names are separated by two dashes. In general, category and branch names must: consist only of letters, numbers, and dashes; must begin with a letter; must not themselves contain two dashes; and must not end with a dash.

Choose a Version Number

Finally, you must choose a version number for the version of `hello-world` that you'll be working on, and create that version in the archive.

Version numbers in `arch` are *not* the name of a particular "snapshot" or release of your project -- though they are related to that concept. Instead, version numbers are the name of a *development line* : a sequence of changes that you make while creating a particular release.

In this case, we'll use the name:

```
        hello-world--mainline--0.1
                               ^^^
                                |
                          version number
```

Notice that version numbers are always positive integers, separated by periods.

Preparing the Archive

Having chosen a name, it's time to prepare the archive for use of that name:

```
        % tla archive-setup hello-world--mainline--0.1
```

After that command, we can query the archive to see what we've done:

```
        % tla categories
        hello-world
```

```
% tla branches hello-world
hello-world--mainline
```

```
% tla versions hello-world--mainline
hello-world--mainline--0.1
```

Why is it Like This

People new to `arch` are sometimes startled at the rigidity of its archive namespace. Two most common question is:

Why have categories, branches and versions? Why can't I just name my projects with arbitrary string? These questions are best answered by recalling that a revision control system is a *librarian*. Part of its job is to help people navigate and search through very large collections of projects and source code. In order to make such searching practical, `arch` defines a *cataloging system*: categories, branches, and versions. (See What is Revision Control?.)

This is somewhat analogous to the cataloging systems used in libraries for books, such as the Dewey decimal classification system: it's a hierarchical categorization of everything in the library. It's a uniform way to describe where a given item is stored, and it aids searching by suggesting the relationships between various items. For example, a branch is likely most closely related to other branches in the same category. A version with a higher major version number most likely contains later work than one in the same branch with a lower major version number.

The analogy isn't perfect: book cataloging systems such as Dewey are based on an official list of categories and subcategories, while `arch` , on the other hand, let's you choose your own category names. Still, like Dewey, `arch` names are based on the idea of grouping related items together to make them easier to search and navigate. And just as Dewey is intended to capture the most common patterns of how people search through books, `arch` is intended to capture the most common patterns of how people search through source archives.

How it Works -- Creating Categories, Branches, and Versions

What does the command `archive-setup actually do? It` s conceptually quite simple: it creates new directories in your archive:

```
        % tla whereis-archive lord@emf.net--2003-example
        /home/lord/{archives}/2003-example

        % cd `tla whereis-archive lord@emf.net--2003-example`
```

Categories are top level directories:

```
        % ls
        =meta-info      hello-world
```

Branches the next level:

```
        % ls hello-world
        hello-world--mainline
```

Versions the third:

```
        % ls hello-world/hello-world--mainline
        hello-world--mainline--0.1
```

Versions are themselves directories:

```
        % ls hello-world/hello-world--mainline/hello-world--mainline--
0.1/
        +revision-lock  +version-lock
```

Note: The lock files (e.g. `+revision-lock`) are used internally by arch. When adding new data to an archive, `arch` doesn't simply call `mkdir` . Instead, it carefully modifies archives to that they are always in a consistent state, regardless of what commands are issued concurrently, or whether or not a command is killed in mid-execution.

Starting a New Source Tree

up: arch Meets hello-world
next: Project Tree Inventories
prev: Starting a New Project

After following the examples in earlier chapters, you should have a new archive and new `hello-world` project within that archive.

In this chapter, we'll walk through the steps of preparing a source tree to be part of that project.

The Intial Source

For the sake of example, let's assume that we have an initial, slightly buggy, implementation of `hello-world` :

```
% cd ~/wd

% ls
hello-world

% cd hello-world

% ls
hw.c    main.c

% cat hw.c

#include <stdio.h>

void
hello_world (void)
{
  (void)printf ("hello warld");
}


% cat main.c

extern void hello_world (void);
```

```
int
main (int argc, char * argv[])
{
  hello_world ();
  return 0;
}
```

Initializing a Project Tree

The first step of preparing source is to turn the ordinary source tree into a *project tree* :

```
% cd ~/wd/hello-world

% tla init-tree hello-world--mainline--0.1

% ls
hw.c    main.c  {arch}
```

Note that we passed `init-tree` the name of the version in the archive that we'll be working on. `init-tree` created a new subdirectory in the root of the tree (`{arch}`).

The `{arch}` subdirectory indicates that this is the root of a project tree:

```
% tla tree-root
/usr/lord/wd/hello-world
```

`tla` knows what archive version this tree is for:

```
% tla tree-version
lord@emf.net--2003-example/hello-world--mainline--0.1
```

Finally, `arch` has created something called a *patch log* for the version passed to `init-tree` :

```
% tla log-versions
lord@emf.net--2003-example/hello-world--mainline--0.1
```

We'll explain what patch logs are for in later chapters.

Initializing a Tree Does Not Change an Archive

So far, we've only marked the project tree as source: we haven't yet stored anything new in the archive. We'll get there, but before we do that, there's an important topic to cover first: *source inventories*. We'll cover that in the next chapter.

What if You Make a Mistake With init-tree?

Suppose that in the example above, we had mis-typed:

```
        % tla init-tree hello-world--mainlin--0.1
```

One "brute force" solution is just to delete the `{arch}` subdirectory and start over. Later on, though, that solution is undesirable: the `{arch}` subdirectory may contain some data you don't want to delete. So, we'll take this opportunity to introduce a few more advanced commands.

There are two problems after the bogus call to `init-tree` . The output from both of these commands is not what we want:

```
        % tla tree-version
        lord@emf.net--2003-example/hello-world--mainlin--0.1

        % tla log-versions
        lord@emf.net--2003-example/hello-world--mainlin--0.1
```

We can change the `tree-version` of a tree at any time:

```
        % tla set-tree-version hello-world--mainline--0.1

        % tla tree-version
        lord@emf.net--2003-example/hello-world--mainline--0.1
```

Patch logs are a little trickier. We have to delete the logs we don't want, and add those that we do want:

```
        % tla add-log-version hello-world--mainline--0.1

        % tla log-versions
        lord@emf.net--2003-example/hello-world--mainlin--0.1
        lord@emf.net--2003-example/hello-world--mainline--0.1

        % tla remove-log-version hello-world--mainlin--0.1

        % tla log-versions
        lord@emf.net--2003-example/hello-world--mainline--0.1
```

WARNING: `remove-log-version` is a dangerous command: it will remove patch logs that you might need if you ask it to. You should only use `remove-log-version` when you are certain, as we were above, that what is being removed is one you do not want.

How it Works -- Initializing a New Tree

`init-tree` created the `{arch}` subdirectory at the root of the source tree. What's in there?

```
        % ls {arch}
        ++default-version      =tagging-method         hello-world

        % cat {arch}/++default-version
        lord@emf.net--2003-example/hello-world--mainline--0.1

        % cat {arch}/=tagging-method
        [... long output ...]
```

`{arch}/hello-world` is the root of a fairly deep tree. Patch logs are stored within that tree.

`{arch}/=tagging-method` is a configuration file that you can use to customize the naming conventions that apply to this tree. It is explained in a later chapter (see Customizing the inventory Naming Conventions).

Note: You should not, of course, edit the contents of the `{arch}` directory by hand.

Project Tree Inventories

up: arch Meets hello-world
next: Inventory Ids for Source
prev: Starting a New Source Tree

Caution: Steep Learning Curve: The concepts and commands introduced in this chapter are likely to be unfamiliar to you, even if you have used other revision control systems. They're really quite simple once you get over the initial learning hurdle -- and after that they're very useful.

The Name-based inventory Concept

In a project tree, some of the files and directories are "part of the source" -- they are of interest to `arch` . Other files and directories may be scratch files, editor back-up files, and temporary or intermediate files generated by programs. Those other files should be ignored or treated specially by most `arch` commands.

This chapter discusses how `arch` recognizes which files to pay attention to, and which to ignore.

The inventory Command

up: Project Tree Inventories
next: The arch Naming Conventions
prev: The Name-based inventory Concept

The command `tla inventory --names --source` is used to print a list of source files as determined by the naming conventions. It has many options, including options to print other kinds of file lists (such as a list of all editor backup files, or a list of all files which are not source).

Let's suppose that after some editing, our source tree looks like this:

```
% ls
hw.c            hw.c.~1~        main.c          {arch}
```

The file `hw.c.~1~` is an editor backup file. `tla` knows that and omits that file from the source inventory:

```
% tla inventory --names --source
./hw.c
./main.c
```

`tla` can give you other lists besides lists of source:

```
% tla inventory --names --backups
./hw.c.~1~
```

The arch Naming Conventions

up: Project Tree Inventories
next: Naming Conventions Illustrated
prev: The inventory Command

This section describes the default naming conventions used by `arch` to pick out source files from other kinds of files. A later chapter describes how to customize these conventions for a partiuclar tree (see Customizing the inventory Naming Conventions).

The naming conventions are based on several categories of files:

```
. and ..                These are simply ignored by arch

excluded                Excluded files are normally omitted
                        from a listing, but if the `--all'
```

	flag is passed to `inventory', then these files are put into one of the categories below and included in the listing.
source	These are apparent source files
precious	These are non-source files that should not be automatically deleted
junk	These are non-source files that may be automatically deleted
backups	These are non-source files that may be automatically deleted, but any program that deletes them should treat them as editor backup files (e.g., keep the oldest and newest of them)
unrecognized	These are files that arch doesn't know how to classify -- they fit none of the naming conventions or that have names that appear to be "suspicious".

The algorithm for classifying files by name has several rules. For each file name, each of these rules is checked in the order listed here until the first rule is reached that classifies the file.

Exclude Dot Files The special files `.` and `..` are always excluded from inventory listings.

Non-portable Names are Unrecognized File names containing whitespace, non-printing characters, or a "globbing character" are always classified as `unrecognized` . The globbing characters are:

```
? [ ] * \
```

Excluded File Test If the `--all` flag is *not* given to `inventory` , the file names matching the pattern for excluded files are dropped from the listing. If the name of a directory is excluded, the entire contents of that directory are skipped. By default, the pattern for excluded files matches control files created by `arch` itself:

```
^(.arch-ids|\{arch\})$
```

Junk File Test All file names reaching this step that begin with two commas (`,,`) are classified as `junk` . Temporary files created by `arch` itself begin with two commas. In addition, any file name matching the junk pattern are classified by `junk` . By default, that pattern matches any name beginning with (at least) one comma:

```
^,.*$
```

Incidentally, that default pattern gives rise to a handy trick. If you need to create a scratch file in a source tree, give it a name that begins with a single comma.

Backup File Test By default, a backup file is any file that reaches this step and matches one of the patterns:

```
^.*(~|\.~[0-9]+~)$
^.*\.bak|\.orig|\.rej|\.original|\.modified|\.reject)$
```

Precious File Test By default, a precious file is any that reaches this step and matches one of the patterns:

```
^\+.*$
^(\.gdbinit|\.#ckpts-lock)$
^(=build\.*|=install\.*)$
^(CVS|CVS\.adm|RCS|RCSLOG|SCCS|TAGS)$
```

Suspicious File Test (Unrecognized) Some file names reaching this step are explicitly treated as `unrecognized` on the presumption that they should probably not be present in a source tree. By default, names ending with any of these extensions are treated as `unrecognized` :

```
.o
.a
.so
.core
```

In addition, the filename `core` is (by default) treated as `unrecognized`).

Source File Test Files reaching this step are compared to the pattern for source files. The default pattern is shown below. You should note that this pattern overlaps that for `excluded` files given above. If the `--all` flag is given to inventory, the `excluded` pattern isn't used, and files that would match it instead "fall through" to later steps of this algorithm.

```
        ^([_=a-zA-Z0-9].*|\.arch-ids|\{arch\}|\.arch-project-tree)$
```

In other words, by default, the `arch` control files and directories are source (if not excluded). Files beginning with letters, numbers, underscore, or an equal sign are source.

Unrecognized Files Any left-over file name reaching this step is treated as `unrecognized` .

Naming Conventions Illustrated

up: Project Tree Inventories
next: Customizing the Naming Conventions
prev: The arch Naming Conventions

Using our example, we can illustrate some of the naming conventions.

Recall that our project tree looks like this:

```
        % ls
        hw.c                hw.c.~1~            main.c              {arch}
```

So the ordinary source listing is:

```
        % tla inventory --names --source
        ./hw.c
        ./main.c
```

And all of the source files (none excluded from the list) is:

```
        % tla inventory --names --source --all
```

```
./hw.c
./main.c
./{arch}/.arch-project-tree
./{arch}/=tagging-method
```

We can include directories in this listing:

```
% tla inventory --names --source --all --both
./hw.c
./main.c
./{arch}
./{arch}/.arch-project-tree
./{arch}/=tagging-method
./{arch}/hello-world
./{arch}/hello-world/hello-world--mainline
[... output trimmed ...]
```

We can also look at some lists of non-source files:

```
% tla inventory --names --backups
./hw.c.~1~
```

The `inventory` command has many options that you may wish to explore.

Customizing the Naming Conventions

up: Project Tree Inventories
next: Why is It Like This -- inventory Naming Conventions
prev: Naming Conventions Illustrated

You can alter the patterns used by `inventory` to classify files. This is explained in a later chapter (see Customizing the inventory Naming Conventions).

Why is It Like This -- inventory Naming Conventions

up: Project Tree Inventories
prev: Customizing the Naming Conventions

Many systems provide naming conventions for recognizing source files but users new to `arch` often wonder why `arch` needs so many categories of files. Recall that `arch` has the categories:

```
excluded
source
precious
junk
backups
unrecognized
```

A rationale for each category is explained here:

excluded is provided simply to keep inventory listings brief in the very common case that `arch` control files are of no particular interest. This is similar to the treatment of "dot files" by `ls` and the `--all` flag to `inventory` is similar to the `-a` flag to `ls` .

source is provides simply so that `arch` can reliably distinguish those files from others. For example, when comparing two source trees, `arch` compares only the files in the category `source` .

precious files are those that `arch` should make an effort to preserve. For example, if `arch` needs to make a copy of a project tree for you, it copies the `precious` files along with the `source` . Suppose, for example, that you are taking notes while working on source. You don't want your file of notes to be mistaken for source, but you also don't want them to be lost. A useful trick is to give the file a `precious` name (e.g. `+notes`).

junk Often when working on a project tree, it's convenient to create "throw-away" files. You might want to compile a quick test program or save, for the moment, the output of some command. When enough of these throw-away files have accumulated, it's handy to be able to get rid of them all-at-once, without having to carefully identify which files to toss, and which to keep. `junk` names are perfect for this. When you create one of these throw-away files, give it name like `,foo` . Later, you can feel confident and safe issuing commands like:

```
% rm ,*

% find . -name ',*' | xargs rm

% tla inventory --junk | xargs rm
```

From arch's perspective, junk files have two important properties. First, when copying a tree, the junk files are *not* copied. Second, it is considered safe for arch to overwrite a junk file. In practice, arch will only ever actually overwrite a junk file if that junk file has a name that begins with `,,` .

backups Editor backup files and the backup files created by programs like `patch` often deserve special treatment. For example, if your editor creates "numbered backups", those are *almost* junk files, but rather than deleting all of them, you might want to delete only some of them.

For arch, what is important is that when copying a tree, backup files should not also be copied. For users, what is hopefully most useful is that using the trick:

```
        % tla inventory --junk | xargs rm
```

will not delete backup files.

unrecognized The appearance in a source tree of a file that doesn't fit any known pattern (or that has a suspicious name) most likely indicates that something has gone wrong. Rather than silently ignoring such files or treating them as `precious` or `junk` , `arch` explicitly flags these exceptions in order to be able to give warnings to users.

Overall, adopting file naming conventions is a discipline that many programmers may not be accustomed to, but it's one I strongly recommend. It's easy to stick to these conventions and tools like `inventory` and `tree-lint` (introduced later) help you to keep your source from get out of control.

Inventory Ids for Source

up: arch Meets hello-world
next: Importing the First Revision
prev: Project Tree Inventories

Caution: Steep Learning Curve: As in the previous chapter, the concepts and commands introduced here are likely to be unfamiliar to you, even if you have used other revision control systems. Once you "get it", though, this will seem quite natural. Best of all, this is the last tricky step before we can start storing project trees in an archive.

Looks Like Source vs Really is Source

In the previous chapter, we saw how to find out which files look like source according to the naming conventions:

```
% tla inventory --names --source
hw.c
main.c
```

In this chapter, there's a new distincition: files which *look* like source according to their names, vs. files which *really are* source.

When you save your project tree in an archive, arch will store the files that *really are* source and ignore the rest. We can ask which files really are source by dropping the `--names` option to `inventory` :

```
% tla inventory --source
[no output]
```

It's a little more interesting if we include arch's own "system files and directories" in the listing:

```
% tla inventory --source --all --both
{arch}
{arch}/.arch-project-tree
{arch}/=tagging-method
{arch}/hello-world
[....]
```

but the thing to note here is that `hw.c` and `main.c` aren't listed. Arch thinks they are source in name only. The next section gives a recipe to fix that, and the sections after that explain what's really going on.

The add Command

We can tell arch that our files really are source, and should really be archived with the project, using the `tla add` command:

```
        % tla add hw.c
        % tla add main.c
```

And now we get a better answer from:

```
        % tla inventory --source
        hw.c
        main.c
```

A related command is `tla delete` :

```
        % tla delete hw.c
```

That doesn't delete the file `hw.c` itself:

```
        % ls
        hw.c                hw.c.~1~            main.c              {arch}
```

but it does remove it from the official list of source:

```
        % tla inventory --source
        main.c
```

For the sake of the examples, we need to put `hw.c` back in the list:

```
        % tla add hw.c
        % tla inventory --source
        hw.c
        main.c
```

Let's take a deeper look at what's going on when you `tla add` files:

Two Names for Every File

In the `arch` world, every source file (and directory) in your project tree has two names: a *file path* and a *inventory id* .

The **file path** of a file is the relative path to the file from the root of the project tree. It describes *where* within a source tree a file is located.

The **inventory id** of a file is a (mostly) arbitrary string that is unique to the file within the tree. The inventory id remains constant even if a file is renamed. So while the file path says where a file is located, the inventory id says which file it is that's stored at that path.

The purpose of `tla add` is to assign an inventory id to a file.

In our example, we can examine the ids:

```
        % tla inventory --source --ids
        hw.c    x_very_long_string
        main.c  x_another_very_long_string
        ^^^^    ^^^^^^^^^^^^^^^^^^^^^^^^^^
         |                |
         |            inventory ids
     file paths
```

Ordinarily, when a file is moved, its file path changes, but its inventory id should remain the same. The `tla move` command helps with this. Suppose that we:

```
        % mv hw.c hello.c
```

we should follow that with:

```
        % tla move hw.c hello.c
```

after which:

```
        % tla inventory --source --ids
        hello.c   x_very_long_string
        main.c    x_another_very_long_string
```

Note that `hello.c` has the same inventory id that `hw.c` used to.

We'll come back to the topic of renames later so, for now, let's put things back where they started:

```
        % mv hello.c hw.c
        % tla move hello.c hw.c
```

Quick Aside -- Adding Directories

The `tla add` command applies to directories, too. If we were to create a new subdirectory in the tree, we should `tla add` it:

```
        % mkdir docs
```

```
        % tla inventory --names --source --both
        docs
        hw.c
        hello.c
```

but

```
        % tla inventory --source --both
        hw.c
        hello.c
```

unless

```
        % tla add docs
```

and then

```
        % tla inventory --source --both
        docs
        hw.c
        hello.c
```

But again, for the sake of our example, we don't need docs. We can just:

```
        % rm -rf docs
```

There isn't a need to `tla delete` a directory that we physically remove.

How it Works -- tla add

What `tla add` does is fairly simple. Note that when we added `hw.c` and `main.c` , a new directory was created:

```
        % ls -a
        .               .arch-ids       hw.c.~1~        {arch}
        ..              hw.c            main.c
```

The `.arch-ids` directory is new:

```
        % ls .arch-ids
        hw.c.id         main.c.id
```

```
        % cat .arch-ids/hw.c.id
        very long string
```

The `*.id` files is where the raw data that determines a file id are stored. The command `tla delete` removes those files. The command `tla move` renames them.

The id for a directory is stored slightly differently. For example, when we created a `docs` subdir and gave it an id with `tla add` , that created a file `docs/.arch-ids/=id` .

Keeping Things Neat and Tidy

The command:

```
        % tla tree-lint
```

is useful for keeping things neat and tidy.

`tree-lint` will tell you of any ids for which the corresponding file does not exist. It will tell you of any files that pass the naming conventions, but for which no explicit id exists.

It will also warn you about files that don't fit the naming conventions.

Inventory Ids -- There's More Than One Way to Do It

In this chapter, you've learned about the basic commands `add` , `move` , and `delete` .

The use of those tools for managing inventory ids was chosen as the default behavior because, superficially at least, it resembles similar commands in systems such as CVS which many users are already familiar with.

There are other ways to manage inventory ids. Sometimes the other ways are more convenient. A later chapter discusses these other techniques (see: xref : *!!!*).

Why is it Like This -- The Purpose of Inventory Ids

As you'll see in later chapters, `arch` is good at managing *changes* made to source trees and the files they contain, and good at telling you about the *history* of trees and files.

As an example, let's suppose that Alice and Bob are both working on the `hello_world` project. In her tree, Alice makes some changes to `hw.c` . In his tree, Bob renames `hw.c` to `hello.c` .

At some point it is necessary to "sync-up" Alice and Bob. Bob should wind up with the changes Alice has been making. Alice should wind up with the same file renaming that Bob has done.

`arch` provides many mechanisms for that syncing up -- it's one of the most important things that `arch` can do -- but nearly all of them boil down to computing and applying **changesets**.

Alice can ask `arch` to create a changeset describing the work she's done, and that changeset will describe the changes she made within `hw.c`. Bob can create a changeset and that changeset will describe the file renaming he did.

If Alice applies Bob's changeset to her tree, her copy of `hw.c` should be renamed `hello.c`. But a trickier case is this: What happens if Bob applies Alice's changeset to his tree?

Alice changed a file named `./hw.c`, but in Bob's tree, those same changes should be made to a file named `./hello.c`. Fortunately, both files have the same inventory id:

```
        file path                inventory id
        ---------                -------------


                 Alice's tree:
        ./hw.c                   x_very_long_string
                                                  \
                                                   - the same long
string
                 Bob's tree:                      /
        ./hello.c                x_very_long_string
```

In Alice's changeset, the changes Alice made are described as being made to the file whose id is `x_very_long_string`.

Therefore, when applying that changeset to Bob's tree, `arch` knows to apply the changes to the file with that same id; it knows to apply the changes to his `./hello.c`.

That example illustrates what inventory ids are for: they allow `arch` to describe the changes made to a tree in terms of the logical identity of files rather than their physical location. There are many more complicated examples of how inventory ids come into play, but now you've seen at least the basic point.

Why is it Like This -- Why tla move Doesn't Move Files

Why doesn't `tla delete` delete the file being removed from the source category, or `tla move` rename it?

Those commands work as they do so that you can adjust the ids in a tree even if some other tool which knows nothing about arch has rearranged files. For example, if you use a "directory editor" to rename source files, `tla move` is available to catch-up to the changes the directory editor made.

Sometimes, arch users request the addition of commands: `tla mv` , `tla mkdir` , `tla rmdir` , and `tla rm` that would modify *both* ids and the corresponding source files. That's a great idea and it's not all that hard: so, if you're looking for something to do, that's a good idea for a real-world programming project on which to try-out and learn arch. Let us know on the `gnu-arch-users` mailing list if you do this, so that we can consider merging your changes into the distribution.

Late Note: One user recently contributed a `tla mv` command which aims to be an inventory-id-aware replacement for `mv(1)` .

Importing the First Revision

up: arch Meets hello-world
next: Checking-in Changes
prev: Inventory Ids for Source

Just to Review: If you've been following the examples in the earlier chapters, we now have:

Your arch User ID In Introducing Yourself to arch, you set an ID string that `arch` uses to identify you.

Your First Archive In Creating a New Archive, you created your first archive and made that your default archive. In Starting a New Project you added the `hello-world` project to that archive.

Your Initial Source Tree In Starting a New Source Tree you began to initialize the sources for `hello-world` as an `arch` project tree and in Inventory Ids for Source you assigned inventory ids to the source files in that project.

Now it's **finally** time to import the sources for `hello-world` into your archive. That will happen in two steps: `(1)` create a log message; `(2)` import the sources.

Making the First Log File

You're about to create a new *revision* of `hello-world` in your archive: a record of how that project looked at a particular point in time.

Whenever you create a new revision, the first step is to create a log file for that revision:

```
% cd ~/wd/hello-world

% tla make-log
++log.hello-world--mainline--0.1--lord@emf.net--2003-example
```

The output from that command is the name of a file which you must now edit. Initially it contains:

```
Summary:
        Keywords:
```

You should fill out this file just like an email message. Add a short description of the revision in the `Summary:` field, and a full description in the body. Just as in email, the body must be separated from the headers by a blank line. When you're done, the log might look like this:

```
        Summary: initial import
        Keywords:

        This is the initial import of `hello-world', the killer app
        that will propel our new .com company to a successful IPO.
```

Usage Note for vi Fans: The default filename of log messages starts with the character `+` . `vi` is a non-standard program in the sense that it treats arguments starting with `+` as options rather than ordinary arguments. Therefore, you should be sure to type the filename for `vi` starting with `./` , as in:

```
    % vi ./++log.hello-world--mainline--0.1--lord@emf.net--2003-
example
```

or you could simply:

```
    % vi `tla make-log`
```

Shortcut Note: This section describes the "long way" to make the log entry to go with your initial import. There is a short-cut that can let you skip this step: the `-L` and `-s` options to `tla import` . We've walked though the long way here but later you might want to try `tla import -H` to learn about the shortcut'.

Storing the First Revision in the Archive

Finally, we can ask `arch` to add our source to the archive:

```
        % tla import
        [....]
```

Note: If you have received an error along the lines of *These apparent source files lack inventory ids* , please reread Inventory Ids for Source and either add each file or change the id-tagging-method to names.

We can observe the side effects of that command in a few ways.

For one thing, we can ask `arch` what revisions exist in the archive for our project:

```
        % tla revisions hello-world--mainline--0.1
        base-0
```

In fact, we can get more detail:

```
        % tla revisions --summary --creator --date \
                          hello-world--mainline--0.1
        base-0
            2003-01-28 00:45:50 GMT
            Tom (testing) Lord <lord@emf.net>
            initial import
```

What's changed in the project tree? Recall that we have something called a *patch log* :

```
        % tla log-versions
        lord@emf.net--2003-example/hello-world--mainline--0.1
```

Now it has an entry:

```
        % tla logs hello-world--mainline--0.1
        base-0

        % tla logs --summary --creator --date \
                       hello-world--mainline--0.1
        base-0
            2003-01-28 00:45:50 GMT
            Tom (testing) Lord <lord@emf.net>
            initial import

        % tla cat-log hello-world--mainline--0.1--base-0
        Revision: hello-world--mainline--0.1--base-0
        Archive: lord@emf.net--2003-example
        Creator: Tom (testing) Lord <lord@emf.net>
        Date: Mon Jan 27 16:45:50 PST 2003
        Standard-date: 2003-01-28 00:45:50 GMT
        Summary: initial import
        Keywords:
        New-files: ./hw.c ./main.c
        New-patches: \
          lord@emf.net--2003-example/hello-world--mainline--0.1--base-
0

        This is the initial import of `hello-world', the killer app
        that will propel our new .com company to a successful IPO.
```

Revision Names from import

`import` created a new revision in the archive. Note that the revision it created is called `base-0` and that we can form a longer name for that revision by prepending the category, branch, and version:

```
        hello-world--mainline--0.1--base-0
        ^^^^^^^^^^^  ^^^^^^^^  ^^^  ^^^^^^
             |          |       |      |
             |          |       |    patch level name
             |          |       |
             |          |     version number
             |          |
             |     branch name
             |
         category name
```

If we add in the archive name, we get something called a *fully qualified revision name* , which is a globally unique identifier for the revision:

```
    lord@emf.net--2003-example/hello-world--mainline--0.1--base-0
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
                |
          archive name
```

Fully qualified names will be of increasing importance as you learn about distributed repositories in later chapters.

How it Works -- What import Does

Let's look at what `import` did to the archive:

```
# cd to the directory for the version we are working
# on:
#
% cd ~/{archives}
% cd 2003-example/
% cd hello-world/
% cd hello-world--mainline/
% cd hello-world--mainline--0.1/
% ls
base-0
```

It created a new `base-0` directory for the revision.

```
% cd base-0
% ls
+revision-lock
hello-world--mainline--0.1--base-0.src.tar.gz
log
```

As always, the `+revision-lock` file is something `arch` uses internally to keep the archive in a consistent state under all circumstances.

The `log` file is a copy of the log message you wrote, with some additional headers added:

```
% cat log
Revision: hello-world--mainline--0.1--base-0
Archive: lord@emf.net--2003-example
Creator: Tom (testing) Lord <lord@emf.net>
Date: Mon Jan 27 16:45:50 PST 2003
Standard-date: 2003-01-28 00:45:50 GMT
Summary: initial import
Keywords:
New-files: ./hw.c ./main.c
New-patches: \
  lord@emf.net--2003-example/hello-world--mainline--0.1--base-0

This is the initial import of `hello-world', the killer app
that will propel our new .com company to a successful IPO.
```

Finally, the compressed tar file is a copy of the source files in your project tree:

```
        % tar ztf hello-world--mainline--0.1--base-0.src.tar.gz
        hello-world--mainline--0.1--base-0/
        hello-world--mainline--0.1--base-0/hw.c
        hello-world--mainline--0.1--base-0/main.c
        hello-world--mainline--0.1--base-0/{arch}/
        hello-world--mainline--0.1--base-0/{arch}/.arch-project-tree
        hello-world--mainline--0.1--base-0/{arch}/=tagging-method
        hello-world--mainline--0.1--base-0/{arch}/hello-world/
        [....]
```

You should notice that the tar file does not include *every* file from your project tree. Specifically, it contains those files that are listed by:

```
        % cd ~/wd/hello-world

        % tla inventory --source --both --all
        [....]
```

Finally, if you poke around in the `{arch}` subdirectory of your project tree, you'll see two new items:

```
        % ls
        ++default-version       =tagging-method
        ++pristine-trees        hello-world
```

The directory `++pristine-trees` contains (at some depth) a copy of the tree you just imported. This is a cached copy used by other `arch` commands. (Note: In future releases of `arch` , it is likely that the `++pristine-trees` subdirectory will be replaced by a different mechanism.)

If you dig around in the `hello-world` (patch log) directory, you can find a local copy of the log file for the revision you just created (with extra headers added to that log file).

Checking-in Changes

up: arch Meets hello-world
next: Retrieving Earlier Revisions
prev: Importing the First Revision

So far, if you're following the examples, we've created a new archive and a `hello-world` project within that archive, and we've imported the initial version of `hello-world` into the archive.

The most common task you're likely to perform as a programming using a revision control system is to *commit* a set of changes. In this chapter, we'll look at the most basic way that that works.

warld != world\n

If you look at our `hello-world` sources, you might notice a spelling error and newline bug:

```
% cat hw.c

#include <stdio.h>

void
hello_world (void)
{
  (void)printf ("hello warld");
}
```

Clearly, we meant to say `hello world` , not `hello warld` and, if we're going to be conventional, we probably wanted a newline at the end of the message. So, let's fix those bugs now.

Some Free Advice About Log Messages

Free advice is worth what you pay for it. -- anonymous.

Here's the plan for fixing these bugs: We'll change the source to fix the bugs. Then we'll ask `arch` to record the changes need to fix the bugs in the archive. That second step will create a new *revision* in the archive.

As we noted earlier, whenever you create a new revision, you need to provide a *log message* for that revision (see Making the First Log File).

The particular bugs we're about to fix in our toy example are quite trivial -- but in a real world situation, they would likely be more complicated. You have a choice: you can either wait until all the changes are done to write the log message describing your changes, or you can write the log message as you go along.

Here's the free advice: write the log message as you go along. In other words, take notes as you hack. In terms of `tla` commands, that means to start the bug fix process with:

```
% cd ~/wd/hello-world

tla make-log
++log.hello-world--mainline--0.1--lord@emf.net--2003-example
```

Then edit your new log file so that it reads:

```
Summary: Fix bugs in the "hello world" string
Keywords:
```

The `Summary:` thus explains what you intend to do with the upcoming changes. As you work, you can fill in the body of the log message.

The Edit/Update-Log Cycle

Pretending that these bugs are more complicated than they actually are, here's how the work might go:

Fix the spelling error. Change `warld` to `world` .

Update the log message. Add a note to the log file:

```
Summary: Fix bugs in the "hello world" string
Keywords:

Spell "world" correctly (not "warld").
```

Fix the newline error. Add a newline to the message.

Update the log message again. Add a note to the log file:

```
Summary: Fix bugs in the "hello world" string
Keywords:

Spell "world" correctly (not "warld").

Add a newline to the hello world message.
```

Oh My Gosh -- What Have I Done?

So you've just worked long and hard on these complex bug fixes. Wouldn't it be a good idea to review your work once more before publishing it?

No problem, `arch` can help:

```
tla changes --diffs
[....]
*** patched regular files

**** ./hw.c
[....]
     @@ -4,7 +4,7 @@
      void
      hello_world (void)
      {
     -  (void)printf ("hello warld");
     +  (void)printf ("hello world\n");
      }
[....]
```

Aha! Now we know. It's time to record that change in the archive.

Storing Changes in the Archive

So now let's record those changes in the archive.

If you didn't take our free advice (see *Some Free Advice About Log Messages*), now is the time to create a log message (hint: `tla make-log`).

To save your changes in the archive, simply:

```
        % tla commit
        [....]
```

After the `commit` completes, there is a new revision in the archive:

```
        % tla revisions hello-world--mainline--0.1
        base-0
        patch-1
```

or in more detail:

```
        % tla revisions --summary hello-world--mainline--0.1
        base-0
            initial import
        patch-1
            Fix bugs in the "hello world" string
```

Our project tree patch log has been similarly updated:

```
        % tla logs hello-world--mainline--0.1
        base-0
        patch-1

        % tla logs --summary hello-world--mainline--0.1
        base-0
            initial import
        patch-1
            Fix bugs in the "hello world" string
```

How it Works -- commit of a New Revision

What does `commit` do to an archive?

```
# cd to the directory for the version we are working
# on:
#
% cd ~/{archives}
% cd 2003-example/
% cd hello-world/
% cd hello-world--mainline/
% cd hello-world--mainline--0.1/
% ls
% ls
+version-lock   =README         base-0          patch-1
```

The `patch-1` subdirectory is new:

```
% cd patch-1

% ls
+revision-lock
hello-world--mainline--0.1--patch-1.patches.tar.gz
log
```

As usual, the log file is the log file you wrote, with some extra headers added:

```
% cat log
Revision: hello-world--mainline--0.1--patch-1
Archive: lord@emf.net--2003-example
Creator: Tom (testing) Lord <lord@emf.net>
Date: Mon Jan 27 22:26:13 PST 2003
Standard-date: 2003-01-28 06:26:13 GMT
Summary: Fix bugs in the "hello world" string
Keywords:
New-files: \
  {arch}/hello-world/ [....] /patch-log/patch-1
Modified-files: hw.c
New-patches: \
          lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-1

        Spell "world" correctly (not "warld").

        Add a newline to the hello world message.
```

The `.patches.tar.gz` file is something called a **changeset**. It describes the changes you made as differences between the `base-0` revision and the `patch-1` revision. You'll learn more about the nature of changesets in later chapters. For now, you can think of a changeset as *similar* to the output of `diff -r` if used to compare the `base-0` revision before your recent changes, with that same tree after your recent changes (or, in the words of one `arch` user: a "patch set on steroids").

In the project tree:

```
        % cd ~/wd/hello-world
```

the commit command had two effects. First, it added a log file under `{arch}/hello-world` . Second, it modified `{arch}/++pristine-trees` to contain a cached copy of the `patch-1` revision instead of the `base-0` revision.

Retrieving Earlier Revisions

up: arch Meets hello-world
next: Shared and Public Archives
prev: Checking-in Changes

If you've followed along with the examples in earlier chapters, you should have:

Your First Archive which is also your default archive:

```
        % tla my-default-archive
        lord@emf.net--2003-example

        % tla whereis-archive lord@emf.net--2003-example
        /usr/lord/examples/{archives}/2003-example
```

A hello-world Project in Your Archive

```
        % tla categories
        hello-world

        % tla branches hello-world
        hello-world--mainline

        % tla versions hello-world--mainline
        hello-world--mainline--0.1
```

Two Revisions of the hello-world Project

```
        % tla revisions hello-world--mainline--0.1
        base-0
        patch-1
```

In this chapter, you'll learn how to retrieve revisions from your archive.

Checking Out the Latest Revision

You might also have a left-over project tree. If so, let's get rid of that:

```
        % cd ~/wd

        % ls
        hello-world

        % rm -rf hello-world
```

Let's suppose that you now want to get the latest sources for the hello world project. For that, you want to use the `get` command:

```
        % tla get hello-world--mainline--0.1 hello-world
        [...]


        % ls
        hello-world

        % ls hello-world
        hw.c    main.c  {arch}
```

Checking Out An Earlier Revision

Let's suppose we want to check out an earlier version of the `hello-world` project.

Notice that in the previous example, we asked just for a particular version of the project:

```
        % tla get hello-world--mainline--0.1 hello-world
                  ^^^^^^^^^^^  ^^^^^^^^  ^^^ ^^^^^^^^^^^
                       |           |      |       |
                       |           |      |  target directory
                       |           |      |
                       |           |      |
                       |           |   version number
```

```
               |           |
               |      branch name
               |
         category name
```

We can get an earlier revision name by specifying its **patch level** explicitly:

```
% tla get hello-world--mainline--0.1--base-0 hello-world-0
          ^^^^^^^^^^^  ^^^^^^^^  ^^^  ^^^^^^ ^^^^^^^^^^^^^
               |           |      |      |        |
               |           |      |      |  target directory
               |           |      |      |
               |           |      | patch level name
               |           |      |
               |           |   version number
               |           |
               |      branch name
               |
         category name


% ls
hello-world     hello-world-0

% ls hello-world-0
hw.c    main.c  {arch}
```

You can see the changes made from `base-0` to `patch-1` with, for example, `diff -r` :

```
% diff -r hello-world-0 hello-world
diff -r hello-world-0/hw.c hello-world/hw.c
7c7
<   (void)printf ("hello warld");
---
>   (void)printf ("hello world\n");
[...]
```

How it Works -- Retrieving Revisions With get

Retrieving the `base-0` revision is easy. As you should recall, the `base-0` revision is stored as a compressed tar file of the complete source tree (see How it Works -- What import Does). When asked to retrieve `base-0` , the `get` command essentially just unpacks that tar file.

Retrieving the `patch-1` revision happens in two steps. Recall that `patch-1` is stored as a *changeset* that describes the differences between `base-0` and `patch-1` (see How it Works -- commit of a New Revision). Therefore, `get` works by first retrieving the `base-0` revision, then retrieving the `patch-1` changeset, then using that changeset to modify the `base-0` tree and turn it into a `patch-1` tree. Internally, `get` uses a tla command called `dopatch` to apply a changeset, but if you are familiar with `diff/patch` patchsets, then you can think of `dopatch` as "patch on steroids".

Let's suppose that instead of committing just one change you'd committed many changes: not just a `patch-1` revision but `patch-2` , `patch-3` and so forth. In essence, `get` will apply each changeset in order to create the revision you requested.

Note: In fact, `get` is a bit more complicated than is described here. On the one hand, there are performance optimizations that can spare `get` from having to apply a long list of changesets. On the other hand, there can be revisions created by `tag` rather than `commit` , for which different rules apply. You'll learn more about these exceptions in later chapters.

Shared and Public Archives

up: arch Meets hello-world
next: The update/commit Style of Cooperation
prev: Retrieving Earlier Revisions

In the earlier chapters, you learned how to create your first archive, start a project, check in the initial sources and subsequent changes, and retrieve past revisions.

In this chapter you'll learn how to make an archive available over a network and begin to learn how multiple programmers can share a single archive.

Registering for Network Access to Archives

As you should recall, an archive has both a logical name, and a physical location:

```
        % tla archives
        lord@emf.net--2003-example
        ^^^^^^^^^^^^^^^^^^^^^^^^^^
          |     /usr/lord/{archives}/2003-example
          |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
          |                      |
          |             archive location
          |
        archive name
```

(See Creating a New Archive.)

Some archives can be accessed over a network, currently via any of the protocols:

```
        FTP
        SFTP
        WebDAV
        plain HTTP
```

Later in this chapter, you'll learn how to create such archives.

For now, you should know that to access such an archive, you register it's name and physical location, using a URL for the physical location.

For example, to access an HTTP or WebDAV archive:

```
    % tla register-archive lord@emf.net--2003b \
            http://regexps.srparish.net/{archives}/lord@emf.net--2003b
```

or an FTP archive:

```
    % tla register-archive lord@regexps.com--2002 \
            ftp://ftp.regexps.com/{archives}/lord@regexps.com--2002
```

You can see that these commands have taken effect:

```
    % tla archives
    lord@emf.net--2003b
            http://regexps.srparish.net/{archives}/lord@emf.net--2003b
    lord@emf.net--2003-example
            /usr/lord/examples/{archives}/2003-example
    lord@regexps.com--2002
            ftp://ftp.regexps.com/{archives}/lord@regexps.com--2002
```

Working with Several Archives at Once

After you've registered additional archives, how do you access them?

One trivial way is to make the archive you are interested in your default:

```
        % tla my-default-archive lord@emf.net--2003

        % tla categories
        [...categories in the remote archive...]
```

It can, of course, be inconvenient to keep changing your default archive. So for now, let's restore it to the archive we've been using in the examples:

```
        % tla my-default-archive lord@emf.net--2003-example
```

There are two other ways to access a remote archive:

Selecting an Archive with -A

Every command that operates on archives accepts a `-A` option which can be used to override the default:

```
        % tla my-default-archive
        lord@emf.net--2003-example

        % tla categories -A lord@emf.net--2003
        [... categories in lord@emf.net--2003 ...]
```

Usage Note: A `-A` argument takes precedence over your default archive but is overridden by fully qualified project names (see below).

Fully Qualified Project Names

Commands that accept project names allow you to use *fully qualified project names* . A fully qualified name is formed by prefixing an archive name, followed by a slash, to the project name:

```
   category name:
   tla                     => lord@emf.net--2003/tla

   branch name:
   tla--devo               => lord@emf.net--2003/arch--tla

   version name:
   tla--devo--1.0          => lord@emf.net--2003/tla--devo--1.0

   revision name:
   tla--devo--1.0--patch-1 => lord@emf.net--2003/tla--devo--1.0--
patch-1
```

As in this example:

```
        % tla my-default-archive
        lord@emf.net--2003-example

        % tla branches lord@emf.net--2003/hello-world
        [... branches of hello-world in lord@emf.net--2003 ...]
```

Usage Note: A fully qualified name takes precedence over both `-A` arguments and your default archive.

Read-only Archives

Operating system and server access controls can be used to limit some or all users to *read-only* access. For example, FTP is usually configured in such a way that anonymous users can read, but not modify the archive.

Creating Local Mirrors, Remote Mirrors, and Remote Archives

A *mirror* is an archive whose contents are copied from another archive. You can not commit to a mirror in the ordinary way, you can only update it's copy of it's source.

There are two primary uses for mirror archives: one is to make a local copy of a remote mirror (so that it's contents can be accessed without going over a network); the other is to make a remote copy of a local archive (so that others can access that copy).

Mirroring a Remote Archive Locally

Let's suppose that, in order to have the fastest possible access to it, or to be able to use it while disconnected, you want to mirror a remote archive locally rather than accessing it over network.

Supposing that you wanted to do this with `lord@emf.net--2003b` , there are three steps (suppose $remote_location is something like http://my.site.com//archives/lord@emf.net--2003b).

First, register the remote archive under a pseudonum, formed by appending `-SOURCE` to it's name:

```
        % tla register-archive lord@emf.net--2003b-SOURCE
$remote_location
```

Second, create your local mirror:

```
        % tla make-archive --mirror-from lord@emf.net--2003b-SOURCE
$local_location
```

That command will, as a side effect, register `lord@emf.net--2003b` as the name of your local mirror.

Finally, copy data from the remote archive:

```
        % tla archive-mirror lord@emf.net--2003b
```

Whenever the remote archive has been added to, you can incrementally update your mirror by repeating the `tla archive-mirror` step.

If you don't want to mirror the entire archive, you can optionally limit the mirror to specific categories, branches, or versions. See tla archive-mirror -H for more.

Mirroring a Local Archive Remotely

Let's suppose that you have a local archive `mine@somewhere.com` , and you'd like to "publish" a mirror of that archive on the Internet so that other people can read from it.

Assuming that you already have `mine@somewhere.com` registered, you can create the remote mirror with:

```
        % tla make-archive --mirror mine@somewhere.com
$remote_location
```

Arch will write directly to $remote_location, so it must be a writeable transport such as sftp, and not something such as standard http.

You can initialize or incrementally update the contents of the remote mirror with:

```
        % tla archive-mirror mine@somewhere.com
```

One common situation for many people is that they are able to install static files as part of a web site, but they can't provide WebDAV access to that web site. Even under those conditions you can still publish an arch archive, though there are two subtleties.

First, when running make-archive, you need to provide an extra flag:

```
        % tla make-archive --listing --mirror mine@somewhere.com \
                        $remote_location
```

The `--listing` flag causes arch to keep `.listing` files up-to-date in the mirror, and that, in turn, allows people to read from the archive using arch over vanilla HTTP (sans WebDAV support).

Second, it _is_ possible for the `.listing` files to fall out of date (for example, if you kill an `archive-mirror` command at just the right time_. If you know or suspect that has occurred, you can repair the archive in question by running archive-fixup as in this example:

```
        % tla archive-fixup mine@somewhere.com-MIRROR
```

Making a Remote Repository

Although mirroring is a common use of remote repositories, it is possible to create remote repositories which are not mirrors, and then to commit to those directly.

One can create a remote repository with a command such as:

```
        % tla make-archive $archive_name $remote_location
```

or, to create a remote repository with .listing files:

```
        % tla make-archive --listing $archive_name $remote_location
```

Mixing Access Modes

There is nothing to prevent you from making a single archive available via multiple access methods. For example, you can register an FTP accessible archive using a local-filesystem location on the machine that contains the FTP directory, but ask other users to register it with an `ftp:` URL.

The update/commit Style of Cooperation

up: arch Meets hello-world
next: Introducing Changesets
prev: Shared and Public Archives

In earlier chapters, you learned how to add a project to an archive, store the initial sources, store changes made to those sources, and retrieve revisions from the archive.

In the previous chapter, you learned how to make an archive network accessible.

This chapter will begin to explore how multiple programmers can share an archive, with each of them making changes to a particular project.

You should take note at the outset that there are really many subtle variations on how programmers can share archives and otherwise cooperate on a given project. We're starting here with one of the very *simplest* techniques.

Alice and Bob Hack main

Let's suppose that Alice and Bob are both working on the `hello-world` project and that they are sharing a single archive. In the examples that follow, we'll play both roles.

For starters, each programmer will need their own project tree:

```
        % cd ~/wd

        % [ ... remove any directories left from earlier examples ...]


        % tla get hello-world--mainline--0.1  hello-world-Alice
        [....]

        % tla get hello-world--mainline--0.1  hello-world-Bob
        [....]
```

Alice's task is to add some legal notices to each file. When she's done (but has not yet used `commit` to write her changes to the archive), the files look this way:

```
% cd ~/wd/hello-world-Alice

% head -3 main.c
/* Copywrong 1998 howdycorp inc.  All rights reversed.*/

extern void hello_world (void);

% head hw.c
/* Copywrong 1998 howdycorp inc.  All rights reversed. */

#include <stdio.h>
```

Bob, meanwhile, has added a much-needed comment to `main` :

```
% cd ~/wd/hello-world-Bob

% cat main.c
extern void hello_world (void);

int
main (int argc, char * argv[])
{
  hello_world ();

  /* Exit with status 0
   */
  return 0;
}
```

Note that the two programmers now have modified versions of `hello-world` , but neither programmer has the other's changes.

Bob commits First

Let's suppose that Bob is the first to try to commit his changes. Just to review, there are two steps.

First, Bob prepares a log message:

```
% cd ~/wd/hello-world-Bob

% tla make-log
++log.hello-world--mainline--0.1--lord@emf.net--2003-example

[Bob edits the log message.]

% cat ++log.hello-world--mainline--0.1--lord@emf.net--2003-example
Summary: commented return from main
Keywords:

Added a comment explaining how the return from `main'
relates to the exit status of the program.
```

Then he calls `commit` :

```
% tla commit
[...]
```

Alice Can Not commit Yet

Now it's Alice's turn:

```
% cd ~/wd/hello-world-Alice

% tla make-log
++log.hello-world--mainline--0.1--lord@emf.net--2003-example

[Alice edits the log message.]

% cat ++log.hello-world--mainline--0.1--lord@emf.net--2003-example
Summary: added copywrong statements
Keywords:

Added copywrong statements to the source files so
that nobody can steal HowdyCorp's code.
```

And then tries to commit:

```
   % tla commit
   commit: tree is not up-to-date
     (missing latest revision is
       lord@emf.net--2003b--2003-example/hello-world--mainline--0.1--
patch-2)
```

The problem here is that Bob's changes have already been stored in the archive, but Alice's tree doesn't reflect those changes.

Studying Why Alice Can Not commit

The `commit` command told Alice that her tree is "out of date". That means that changes have been committed to the archive that her tree doesn't have yet.

She can examine the situation in a little more depth by asking what her tree is missing:

```
        % tla missing
        patch-2
```

or for more detail:

```
        % tla missing --summary
        patch-2
            commented return from main
```

which you should recognize as the `Summary:` line from Bob's log message.

She can get even more detail with the (previously introduced) `revisions` command (see Storing the First Revision in the Archive).

She can view Bob's entire log message:

```
    % tla cat-archive-log hello-world--mainline--0.1--patch-2
    Revision: hello-world--mainline--0.1--patch-2
    Archive: lord@emf.net--2003-example
    Creator: Tom (testing) Lord <lord@emf.net>
```

```
Date: Wed Jan 29 12:46:50 PST 2003
Standard-date: 2003-01-29 20:46:50 GMT
Summary: commented return from main
Keywords:
New-files: {arch}/hello-world/[....]
Modified-files: main.c
New-patches: \
  lord@emf.net--2003-example/hello-world--mainline--0.1--patch-2

Added a comment explaining how the return from `main'
relates to the exit status of the program.
```

By looking at the headers of that message, Alice can figure out, for example, that Bob modified the file `main.c`.

In later chapters, we'll explore more commands that Alice can use to study the changes that Bob made, but for now, let's turn to how Alice can add those changes to her tree.

The update Command

Alice needs to combine her changes with Bob's before she can `commit` her changes. One easy way to do that is the `update` command:

```
        % cd ~/wd

        % tla update --in-place hello-world-Alice
        [....]
```

Now she will find Bob's changes added to her tree:

```
        % cd hello-world-Alice

        % cat main.c
        /* Copywrong 1998 howdycorp inc.  All rights reversed. */

        extern void hello_world (void);

        int
```

```
main (int argc, char * argv[])
{
  hello_world ();

  /* Exit with status 0
   */
  return 0;
}

/* arch-tag: main module of the hello-world project
 */
```

Since no further changes are missing:

```
% tla missing
[no output]
```

`commit` is happy to proceed:

```
% tla commit
[....]
```

Learning Note: If you're following along with the examples, you should still have a tree in `hello-world-Bob` that has Bob's changes, but not Alice's. Try various commands for that directory to explore (`missing` , `update` , `changes` and so forth).

How it Works -- The update Command

A *full* explanation of how `update` works is a little beyond the scope of this chapter. You'll be able understand `update` in detail after a few of the later chapters (on changesets and patch logs).

For now, if you are familiar with `diff` and `patch` , you can think of it this way:

When `update` is run in Alice's tree, it notices that the archive is up to a `patch-2` revision, but that her tree was checked out as a `get` of the `patch-1` revision. `update` works in three steps:

First, it uses a command called `mkpatch` (which is kind of a fancier variation on `diff`) to compute a changeset (a fancy patch set) that describes the changes Alice made to her tree.

Second, it checks out a copy of the `patch-2` revision and replaces Alice's tree with that revision.

Third, `update` uses `dopatch` (a fancier `patch`) to apply the changeset from the first step to the new tree.

You may be wondering how patch conflicts are handled. The examples above were carefully crafted to avoid any conflicts. Don't worry -- we'll get to that topic soon enough (see Inexact Patching -- How Conflicts are Handled).

Introducing Changesets

up: arch Meets hello-world
next: Exploring Changesets
prev: The update/commit Style of Cooperation

It is often extremely useful to compare two project trees (usually for the same project) and figure out exactly what has changed between them. A record of such changes is called a *changeset* or a *delta* .

Changesets are a very central concept to `arch` -- much of `arch` is defined in terms of operations performed with changesets.

If you have a changeset between an "old tree" and a "new tree", you can "apply the changeset" to the old tree to get the new tree -- in other words, you can automatically make the editing changes described by a changeset. If you have some third tree, you can apply the patch to get an approximation of making the same changes to that third tree.

`arch` includes sophisticated tools for creating and applying changesets.

- mkpatch
- dopatch
- Inexact Patching -- How Conflicts are Handled

---

mkpatch

up: Introducing Changesets
next: dopatch

`mkpatch` computes a changeset describing the differences between two trees. The basic command syntax is:

```
        % tla mkpatch ORIGINAL MODIFIED DESTINATION
```

which compares the trees `ORIGINAL` and `MODIFIED` .

`mkpatch` creates a new directory, `DESTINATION` , and stores the changeset there.

When `mkpatch` compares trees, it uses inventory ids. For example, it considers two directories or two files to be "the same directory (or file)" if they have the same id -- regardless of where each is located in its respective tree. (See Inventory Ids for Source.)

A changeset produced by `mkpatch` describes what files and directories have been added or removed, which have been renamed, which files have been changed (and how they have been changed), and what file permissions have changed (and how). When regular text files are compared, `mkpatch` produces a context diff describing the differences. `mkpatch` can compare binary files (saving complete copies of the old and new versions if they differ) and symbolic links (saving the old and new link targets, if they differ).

A detailed description of the format of a changeset is provided in an appendix (see The arch Changeset Format).

---

dopatch

up: Introducing Changesets
next: Inexact Patching -- How Conflicts are Handled
prev: mkpatch

`dopatch` is used to apply a changeset to tree:

```
        % tla dopatch PATCH-SET TREE
```

If `tree` is exactly the same as the the "original" tree seen by `mkpatch` , then the effect is to modify `tree` so that it is exactly the same as the the "modified" tree seen by `mkpatch` , with one exception (explained below).

"Exactly the same" means that the directory structure is the same, symbolic link targets are the same, the contents of regular files are the same, and file permissions are the same. Modification times, files with multiple (hard) links, and file ownership are not reliably preserved.

The exception to the "exactly the same" rule is that if the patch requires that files or directories be removed from `tree` , those files and directories will be saved in a subdirectory of `tree` with an eye-splitting name matching the pattern:

```
        ++removed-by-dopatch-PATCH--DATE
```

where `PATCH` is the name of the patch-set directory and `DATE` a timestamp.

Inexact Patching -- How Conflicts are Handled

up: Introducing Changesets
prev: dopatch

What if a tree patched by `dopatch` is not exactly the same as the original tree seen by `mkpatch` ?

Below is a brief description of what to expect. Complete documentation of the `dopatch` process is included with the source code.

`dopatch` takes an inventory of the tree being patched. It uses inventory ids to decide which files and directories expected by the changeset are present or missing from the tree, and to figure out where each file and directory is located in the tree.

Simple Patches If the changeset contains an ordinary patch or metadata patch for a link, directory or file, and that file is present in the tree, `dopatch` applies the patch in the ordinary way. If the patch applies cleanly, the modified file, link, or directory is left in place.

If a simple patch fails to apply cleanly, `dopatch` will always leave behind a `.orig` file (the file originally in the tree being patched, without any changes) and a `.rej` file (the part of the patch that could not be applied).

If the patch was a context diff, `dopatch` will also leave behind the file itself -- partially patched.

If an (unsuccessful) patch was for a binary file, no partially-patched file will be left. Instead, there will be:

```
        .orig   -- the file originally in the tree being patched,
                   without modifications.

        .rej    -- a complete copy of the file from the modified tree,
                   with permissions copied from `.orig'.

        .patch-orig -- a complete copy of the file from the original
```

```
                       tree seen by `mkpatch', with permissions
                       retained from that original

                        -or-

                       the symbolic link from the original tree seen
                       by `mkpatch' with permissions as in the
original
                       tree.
```

If an (unsuccessful) patch was for a symbolic link, no partially patched file will be left. Instead there will be:

```
        .orig   -- the unmodified file from the original tree

        .rej    -- a symbolic link with the target intended by the
                   patch and permissions copied from .orig

        .patch-orig -- a complete copy of the file from the original
                       tree seen by `mkpatch', with permissions
                       retained from that original

                        -or-

                       the symbolic link from the original tree seen
                       by `mkpatch' with permissions as in the
original
                       tree.
```

Patches for Missing Files

All patches for missing files and directories are stored in a subdirectory of the root of the tree being patched called

```
        ==missing-file-patches-PATCH-DATE
```

where `PATCH` is the basename of the changeset directory and `DATE` a time-stamp.

Directory Rearrangements and New Directories

Directories are added, deleted, and rearranged much as you would expect, even if you don't know it's what you'd expect.

Suppose that when `mkpatch` was called the `ORIGINAL` tree had:

```
        Directory or file:              Id:

        a/x.c                           id_1
        a/bar.c                         id_2
```

but the `MODIFIED` tree had:

```
        a/x.c                           id_1
        a/y.c                           id_2
```

with changes to both files. The patch will want to rename the file with id `id_2` to `y.c` , and change the contents of the files with ids `id_1` and `id_2` .

Suppose, for example, that you have a tree with:

```
        a/foo.c                         id_1
        a/zip.c                         id_2
```

and the you apply the patch to that tree. After the patch, you'll be left with:

```
        a/foo.c                         id_1
        a/y.c (was zip.c)               id_2
```

with patches made to the contents of both files.

Here's a sample of some subtleties and ways of handling conflicts:

Suppose that the original tree seen by mkpatch has:

```
        Directory or file:              Id:

        ./a                             id_a
        ./a/b                           id_b
        ./a/b/c                         id_c
```

and that the modified directory has:

```
        ./a                             id_a
        ./a/c                           id_c
        ./a/c/b                         id_b
```

Finally, suppose that the tree has:

```
        ./x                             id_a
        ./x/b                           id_b
        ./x/c                           id_new_directory
        ./x/c/b                         id_different_file_named_b
        ./x/c/q                         id_c
```

When patch gets done with the tree, it will have:

```
        ./x                             id_a
                Since the patch doesn't do anything
                to change the directory with id_a.

        ./x/c.orig                      id_new_directory
        ./x/c.rej                       id_c
                Since the patch wants to make the
                directory with id_c a subdirectory named "c"
                of the directory with id_a, but the tree
                already had a different directory there,
                with the id id_new_directory.

        ./x/c.rej/b                     id_b
                Since the patch wants to rename the directory
                with id_b to be a subdirectory named "b"
                of the directory with id_c.

        ./x/c.orig/b                    id_different_file_named_b
                Since the patch made new changes to this file,
                it stayed with its parent directory.
```

Exploring Changesets

up: arch Meets hello-world
next: Introducing replay -- An Alternative to update
prev: Introducing Changesets

The previous chapter introduced *changesets* and the commands `mkpatch` and `dopatch` (fancy variations on the theme of the traditional `diff` and `patch` programs).

In this chapter, we'll look in a bit more detail about how changesets are used in archives, how they are used by the `commit` and `update` commands, and what this implies for how you can make the best use of `arch` .

How it Works -- commit Stores a Changeset in the Archive

Suppose that you `get` the latest revision of a project, make some changes, write a log message, and `commit` those changes to an archive. What happens?

In essence, commit:

1 Computes a changeset that describes what changes you've made compared to the latest revision.

2 Creates a directory for the new revision in the archive.

3 Stores your log message and the changeset in the archive.

In that light, you might want to go back and review an earlier section: How it Works -- commit of a New Revision.

get-changeset Retrieves a Changeset from an Archive

Earlier, you learned that the `cat-archive-log` command retrieves a log message from an archive (see Studying Why Alice Can Not commit).

You can also retrieve a changeset from an archive:

```
        % cd ~/wd
        % tla get-changeset hello-world--mainline--0.1--patch-1 patch-1
        [...]
```

`get-changeset` retrieves the changeset from the archive and, in this case, stores it in a directory called `patch-1` .

(The format of changesets is described in The arch Changeset Format.)

Using show-changeset to Examine a Changeset

The changeset format is optimized for use by programs, not people. It's awkward to look at a changeset "by hand". Instead, you may wish to consider getting a report of the patch in diff format by using:

```
        % tla show-changeset --diffs patch-1


        [...]
```

If you've been following along with the examples, you'll recognize the output format of `show-changeset` from the `changes` command introduced earlier (see Oh My Gosh -- What Have I Done?).

Using commit Well -- The Idea of a Clean Changeset

When you commit a set of changes, it is generally "best practice" to make sure you are creating a *clean changeset* .

A clean changeset is one that contains only changes that are all related and for a single purpose. For example, if you have several bugs to fix, or several features to add, try not to mix those changes up in a single `commit` .

There are many advantages to clean changesets but foremost among them are:

Easier Review It is easy for someone to understand a changeset if it is only trying to do one thing.

Easier Merging As we'll learn in later chapters, there are circumstances in which you'll want to look at a collection of changesets in an archive and pick-and-choose among them. Perhaps you want to grab "bug fix A" but not "new feature B". If each changeset has only one purpose, that kind of *cherrypicking* is much more practical.

Introducing replay -- An Alternative to update

up: arch Meets hello-world
next: Selected Files Commit
prev: Exploring Changesets

`update` isn't the only way to catch-up with a development path. Another option is `replay` :

```
% cd ~/wd/project-tree
% tla replay
[....]
```

What does that actually do?

An update Refresher

Let's suppose that we check out an old version of `hello-world` :

```
% cd ~/wd
% tla get hello-world--mainline--0.1--patch-1 hw-patch-1
[...]
```

It's easy to see that the resulting tree is not up-to-date:

```
% cd hw-patch-1
% tla missing
patch-2
patch-3
```

Now, let's suppose that we make some local changes in `hw-patch-1` and then run `update` . What happens?

Local changes are computed against patch-1. In other words, a changeset is created that represents the changes from a pristine copy of the `patch-1` revision to the current state of the project tree (`hw-patch-1`).

A copy of patch-3 is checked out. `update` starts with a pristine copy of the `patch-3` revision.

The changeset is applied to the patch-3 tree. The changes computed in the first step are made to the new tree.

There's another way, though:

The replay Command

We have a local copy of the `patch-1` , perhaps with some local changes:

```
% cd ~/wd/hw-patch-1
% tla missing
patch-2
patch-3
```

Recall that the `patch-2` and `patch-3` revisions each correspond to a specific changeset, stored in the archive (see How it Works -- commit of a New Revision).

We could add those changes to your local tree by using `get-changeset` to retrieve each changeset, and `dopatch` to apply it (see get-changeset Retrieves a Changeset from an Archive, and dopatch). That's a lot of tedious work, though, so `arch` provides a more automated way to accomplish that same effect:

```
% cd ~/wd/hw-patch-1
% tla replay
[....]
% tla missing
[no output]
```

`replay` will do just what we've described: get patches from the archive and apply them one-by-one. One word of caution, though: if one of those patches generates conflicts, `replay` will stop there and let you fix the conflicts. You can then pick up where `replay` left off by running `replay` a second time.

How it Works -- replay

If you've followed along with the tutorial so far, the way that `replay` works should be pretty obvious. In fact, it's just exactly how we described it above. `replay` uses `missing` to find out what changes your tree is missing, `get-changeset` to retrieve those changesets, and `dopatch` to apply them.

There's a fair amount of "bookkeeping" involved in doing that -- and that bookkeeping is what `replay` automates for you.

Selected Files Commit

up: arch Meets hello-world
next: Elementary Branches -- Maintaining Private Changes
prev: Introducing replay -- An Alternative to update

Earlier, you learned how to commit all of the changes within a tree at once (see Checking-in Changes).

You also have read a bit about the importance of making "clean" changesets (see Using commit Well -- The Idea of a Clean Changeset).

This chapter shows you a little trick that you can use in a very specific but common situation.

The Quick Fix Problem

Let's suppose that you have a large project tree and you're in the middle of making some complicated change. You've modified many files, but there are many others that you haven't touched.

Suddenly, you notice a trivial bug **in one of the untouched files**.

What you'd really like to do is:

1) Stop and fix the trivial bug.

2) Commit just that trivial bug fix.

3) Get back to work on the complicated changes.

How can you do that?

The Brute Force Solution to the Quick Fix Problem

There's an easy "brute force" solution to the problem.

Simply:

Check out a fresh copy of the latest revision. In other words, create a second project tree with no modifications.

Fix the trivial bug in the new tree and commit. Now you've committed a clean change with just the trivial bug fix.

Use update or replay to Bring Your Original Tree Up to Date. That will add the trivial bug fix back to your tree with the partially completed changes.

That works, but it can be a little awkward. Do you *really* need to start a second project tree just to fix this trivial bug?

Sometimes the awkwardness is well worth it. For example, your project might have a policy the before every `commit` , you must run some tests. In that case, yes, you really do need a second tree.

Sometimes the awkwardness is nearly unavoidable. For example, if the trivial bug fix involves modifying files that you've already heavily modified, then again, the brute force technique may be the simplest approach (but also, take a look at `tla undo --help` and `tla redo --help`).

But there is a simpler way that sometimes applies:

Solving the Quick Fix Problem with commit --

As it turns out, `commit` lets you commit only the changes made to just a few files.

If your quick fix changes `file-a.c` and `file-b.c` , then after preparing a log message, you can commit just those files with:

```
        % tla commit -- file-a.c file-b.c
```

You should note that the files committed this way must not be new files and that, even if those files have been renamed, the `commit` will record only the changes internal to those files, not the renames.

The Quick Fix Problem -- There's More Than One Way to Do It

In the text above, we speculated about a "brute force" solution to the quick-fix problem that involved checking out a whole new project tree.

Two other command, `tla undo` and `tla redo` , provide an alternative "brute force" solution with some advantages. These are described in a later chapter (see xref : *!!!*)../

Elementary Branches -- Maintaining Private Changes

up: arch Meets hello-world
next: Patch Logs and Project Tree History
prev: Selected Files Commit

In this chapter, we'll begin to explore the concept of *branching* , which you may be familiar with from other revision control systems.

If you are already familiar with the concept, you should be aware that branching in `arch` almost certainly goes far beyond what you are accustomed to.

Regardless of whether or not you are familiar with the concept, fear not -- we'll be starting slow:

A Branching Scenario -- The Need for Private Changes

Let's suppose for the moment that the `hello-world` project is making its sources available as a public, read-only mirror (see Shared and Public Archives).

Early on, you (someone not involved in the `hello-world` project) decides that you'll want to use their program, but that you'll need to make some local changes.

As a sort of toy example, let's suppose that you've decided that in your environment, saying *hello world* is unacceptable -- you really require the more correctly punctuated *hello, world* .

Now, here's the problem: sure, you can download their sources and make that change. But meanwhile, the project is going to keep working. They're going to keep making changes. So, you'll be faced with a perpetual task of repeatedly downloading their latest sources and copying your changes to their latest version.

`arch` can help automate that task, and this chapter explains how.

Making a Branch from a Remote Project in a Local Archive

In the examples that follow, you'll be changing roles. Instead of "playing" Alice or Bob, the programmers on the `hello-world` project, you'll be playing Candice: a third party.

Let's start by giving Candice her own archive to use, and making that the default archive:

```
        % tla make-archive candice@candice.net--2003-candice \
                             ~/{archives}/2003-candice

        % tla my-default-archive candice@candice.net--2003-candice
        default archive set (candice@candice.net--2003-candice)
```

(You can review what those commands do by reading Creating a New Archive.)

Candice needs to create a `hello-world` project in her own archive. She can use:

```
        % tla archive-setup  hello-world--candice--0.1
```

She doesn't *have* to use the same project name that Alice and Bob are using and, in fact, in this case she chose a different branch name. (To review those commands, see Starting a New Project.)

When Alice and Bob created their archive, they used `import` to create the first revision. Since we're creating a *branch*, we'll use a different command.

For the sake of example, let's suppose Candice is going to start from the `patch-1` revision of Alice and Bob's archive:

```
    % tla tag \
        lord@emf.net--2003-example/hello-world--mainline--0.1--patch-
1 \
        hello-world--candice--0.1
    [....]
```

There are a few things worth noting about that command.

First, note that we used a **fully qualified revision name** to refer to Alice and Bob's `patch-1` revision. That's because that revision is in some archive other than the current default archive. (See Working with Several Archives at Once.)

Next, note that we specified the `patch-1` revision explicitly. If we had left of the `--patch-1` suffix, then the `tag` command would assume we meant the *latest* revision in Alice and Bob's archive (which happens to be `patch-3`).

What tag Just Did

After using `tag` , Candice now has a new revision in her archive:

```
   % tla revisions --summary hello-world--candice--0.1
   base-0
       tag of lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-1
```

She can retrieve that revision in the usual way:

```
   % tla get hello-world--candice--0.1 hw-candice
   [...]

   % ls hw-candice
   hw.c         main.c          {arch}
```

Nifty arch Feature: If you've followed along closely, you should have noticed that Candice created a branch in her archive from an arch revision stored in another archive entirely. In our examples, both of these archives happen to be on the local file system but that isn't necessary: Candice could have formed her branch even if she was accessing Alice and Bob's archive over the network.

Usage Caution: Candice's job isn't quite done yet. The next section explains another step she'll *probably* want to take.

Caching a tag Revision

Candice used `tag` to create a branch from Alice and Bob's archive. When she uses `get` to check-out that revision, what happens? Roughly speaking, `arch` notices that the revision is a branch, then consults Alice and Bob's archive to really get the source.

The question then arises: what if Alice and Bob's archive "goes away"? As things stand, if that happens, Candice will no longer be able to `get` from her branch.

She can fix that though by caching in her archive all of the information needed to build the revision:

```
% tla cacherev hello-world--candice--0.1--base-0
[...]
```

and confirm that that worked with:

```
% tla cachedrevs hello-world--candice--0.1
hello-world--candice--0.1--base-0
```

Thereafter, `arch` will no longer rely on Alice and Bob's archive to retrieve Candice's `base-0` revision.

Exploring the New Branch

Earlier, Candice created her branch and used `get` to check it out. Let's examine that tree:

```
% cd ~/wd/hw-candice

% tla log-versions
candice@candice.net--2003-candice/hello-world--candice--0.1
lord@emf.net--2003-example/hello-world--mainline--0.1
```

Note that Candice's tree has patch logs both for Alice and Bob's versions, and for her own branch:

```
% tla logs --summary \
        lord@emf.net--2003-example/hello-world--mainline--0.1
base-0
    initial import
```

```
patch-1
    Fix bugs in the "hello world" string


% tla logs --summary hello-world--candice--0.1
base-0
    tag of \
    lord@emf.net--2003-example/hello-world--mainline--0.1--patch-1
```

There are not any later changes on Candice's branch:

```
% tla missing hello-world--candice--0.1
[no output]
```

but recall that Alice and Bob are already up to `patch-3` :

```
% tla missing -A lord@emf.net--2003-example \
        hello-world--mainline--0.1
patch-2
patch-3
```

Making a Local Change

After the initial `tag` , Candice can commit changes to her branch in the usual way.

Let's suppose that she has edited `hw.c` so that it now reads (in part):

```
% cat hw.c
[...]
void
hello_world (void)
{
  (void)printf ("hello, world\n");
}
[...]
```

and that's she's prepared a log message:

```
% cat ++log.hello-world--candice--0.1--lord@emf.net--2003-candice
Summary: Punctuated the output correctly
Keywords:


This program should say "hello, world" not "hello world".
```

Now she can simply commit in the usual way, creating her own `patch-1` revision:

```
% tla commit
[....]

% tla revisions --summary hello-world--candice--0.1
base-0
    tag of \
    lord@emf.net--2003-example/hello-world--mainline--0.1--patch-1
patch-1
    Punctuated the output correctly
```

Updating from a Branched-from Version

Meanwhile, Alice and Bob have gone on to create their revisions `patch-2` and `patch-3` . How can Candice add those changes to her branch?

Well, really, `arch` provides lots of techniques. Using commands we've already introduced, she could use either `update` or `replay` . In this example, we'll demonstrate using `replay` .

```
% cd ~/wd/hw-candice

% tla replay -A lord@emf.net--2003-example \
        hello-world--mainline--0.1
        [...]
```

Note that we used a `-A` argument to say which archive we are replaying changes from, and a version name to say which changes we want. In this case, `replay` applied the changesets for `patch-2` and `patch-3` to Candice's tree.

This use of `replay` is a form of *merging* : Candice's local changes have been merged with Alice and Bob's `mainline` changes.

Learning Note: If you're following along with the examples, you should examine `hw.c` and notice that Candice's change to the `printf` string and Alice's addition of a "copywrong" notice are both included.

Learning Note: You should also check out a second copy of Candice's `patch-1` revision and experiment with doing the same merge using `update` instead of `replay` . You might have to look at `tla update -help` to figure out exactly what options and arguments to provide.

Note also that, so far, we've only made these changes to Candice's project tree -- they haven't been checked into Candice's archive. To actually record the merge in her archive, she'll have to make a log message and commit in the usual way (see Checking-in Changes).

There is, however, one more convenience to point out. When Candice writes her log message, she'll presumably want to note that the merge took place and what it involves. `arch` includes a command whose output is ideal for inclusion in such a log message:

```
  % cd ~/wd/hw-candice

  % tla log-for-merge
  Patches applied:

    * lord@emf.net--2003-example/hello-world--mainline--0.1--patch-3
       added copywrong statements

    * lord@emf.net--2003-example/hello-world--mainline--0.1--patch-2
       commented return from main
```

How It Works -- tag and Elementary Branches

What did `tag` do? Let's look at Candice's archive:

```
% cd ~/{archives}
% cd 2003-candice
% cd hello-world
% cd hello-world--candice
% cd hello-world--candice--0.1

% ls
+version-lock   base-0          patch-1
patch-2
```

Of particular interest is the `base-0` revision -- the one created by `tag` :

```
% cd base-0

% ls
CONTINUATION
hello-world--candice--0.1--base-0.patches.tar.gz
hello-world--candice--0.1--base-0.tar.gz
log

% cat CONTINUATION
lord@emf.net--2003-example/hello-world--mainline--0.1--patch-1
```

The file `CONTINUATION` identifies this revision as a `tag` revision. Its contents tell us what revision we branched from.

The changeset for this revision (`....patches.tar.gz`) was also created by `tag` . If you explore that changeset (recall `get-changeset` and `show-changeset`) you'll see that all it does is add a log entry to the tree's patch log.

The source file (`...base-0.tar.gz`) was created by `archive-cache-revision` . It contains a complete copy of Candice's `base-0` revision. Since that file is there, `get` is not obligated to look at Alice and Bob's archive to construct this revision.

Patch Logs and Project Tree History

up: arch Meets hello-world
next: Development Branches -- The star-merge Style of Cooperation
prev: Elementary Branches -- Maintaining Private Changes

In the previous chapter, we began to learn about branching and merging. We saw how commands like `missing` , `update` , and `replay` can be used to keep track of and apply changes from multiple branches of a project.

In this chapter, we'll explain a bit about *patch logs* : the mechanism that is used to keep track of the history of a project tree, including that part of the history that is used for intelligent merging.

You should recall first encountering patch logs in earlier chapters (for example, when first initializing a project tree, in Starting a New Source Tree). In this chapter, patch logs are explained in greater depth.

Project Trees Have Patch Logs

Recall that every initial import, tag revision, and changeset revision in an archive has an associated log message. That message consists of the headers and body that you supply to commands such as `import` and `commit` , plus additional headers that are automatically generated by `arch` .

When a project tree is first imported to an archive, the patch log entry for the new revision is added to the tree. When a `commit` takes place, as part of the process of committing, the log entry for the new revision is added to the tree. If you `get` a revision created by the `tag` command, you'll also find that it contains a patch log entry for the tag revision.

Patch log entries *accumulate*. Thus, for example, each commit adds a new log entry and all earlier log entries are preserved. Each tag revision includes not only the entry for the tag, but all log entries inherited from the revision being tagged.

Returning to our earlier examples, let's take a look at Alice and Bob's `patch-2` revision:

```
% cd ~/wd

[... remove directories from earlier examples ...]

% tla get -A lord@emf.net--2003-example \
            hello-world--mainline--0.1--patch-2 \
            hw-AnB-2

[...]

% cd ~/hw-AnB-2
```

First, we note that patch logs are sorted by `arch` version names. This tree has logs from only one version:

```
% tla log-versions
lord@emf.net--2003-example/hello-world--mainline--0.1
```

Within that version, it has logs for the initial import, and two changesets:

```
% tla logs -A lord@emf.net--2003-example \
               --summary \
               hello-world--mainline--0.1
base-0
    initial import
patch-1
    Fix bugs in the "hello world" string
patch-2
    commented return from main
```

Examining one of those log entries in particular:

```
% tla cat-log -A lord@emf.net--2003-example \
                hello-world--mainline--0.1--patch-2
Revision: hello-world--mainline--0.1--patch-2
Archive: lord@emf.net--2003-example
Creator: Tom (testing) Lord <lord@emf.net>
Date: Wed Jan 29 12:46:50 PST 2003
```

```
        Standard-date: 2003-01-29 20:46:50 GMT
        Summary: commented return from main
        Keywords:
        New-files: \
          {arch}/[...]/hello-world--mainline--0.1/[...]/patch-log/
patch-2
        Modified-files: main.c
        New-patches: \
          lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-2

        Added a comment explaining how the return from `main'
        relates to the exit status of the program.
```

we can see, for example, that the `patch-2` changeset modified the file `main.c` and added a new file, the log entry itself (whose name is abbreviated in the output displayed above).

Other examples worth considering come from Candice's tree. Recall that she used `tag` to fork from Alice and Bob's tree at their `patch-1` revision. Therefore we see:

```
        % cd ~/wd

        % tla get -A candice@candice.net--2003-candice \
                    hello-world--candice--0.1--patch-2 \
                    hw-C-0

        [...]

        % cd ~/hw-C-0

        % tla log-versions
        candice@candice.net--2003-candice/hello-world--candice--0.1
        lord@emf.net--2003-example/hello-world--mainline--0.1

        % tla logs  -A lord@emf.net--2003-example \
                        --summary \
                        hello-world--mainline--0.1
        base-0
            initial import
        patch-1
            Fix bugs in the "hello world" string


        % tla logs  -A candice@candice.net--2003-candice \
                        --summary \
                        hello-world--candice--0.1
        base-0
            tag of \
              lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-1
```

How It Works -- missing

In earlier chapters, you learned how the command `missing` can tell you about changes commited to archives, but not yet present in a given project tree (see Studying Why Alice Can Not commit and Exploring the New Branch).

It should now be easy to understand how those commands work. `arch` can find the list of all revisions in a given version using the `revisions` command:

```
        % tla revisions -A lord@emf.net--2003-example \
                          hello-world--mainline--0.1
        base-0
        patch-1
        patch-2
        patch-3
```

Those are the logs in the archive. `arch` can find out the list of revisions for which a project tree has log entries with `logs` :

```
        % tla logs -A lord@emf.net--2003-example \
                       hello-world--mainline--0.1
        base-0
        patch-1
        patch-2
```

The difference between those two lists is the output of `missing` :

```
% tla missing -A lord@emf.net--2003-example \
              hello-world--mainline--0.1
patch-3
```

The Concept of Change History and Tree Ancestry

Patch logs give important insight into the history of a tree. There are two views worth mentioning: the *change history* view, and the *tree ancestry* view.

Change History

When a tree has a log for a given `commit` changeset, that means that the changes from that `commit` have been applied to the tree: the `commit` changeset is part of the "change history" of the tree. If the changeset were a bug fix, for example, then this is a likely indication that the bug fix is present in the tree.

Note: The mere fact that a given changeset is part of the change history of a tree isn't absolute proof that the changes made by that changeset are present in the tree. For example, those changes might have been "undone" by a later change. Nevertheless, the change history of a tree is a useful tool for exploring and understanding its state.

Tree Ancestry

Informally, we say that an archived revision is a *tree ancestor* of a given project tree if it has patch log entries for all of the revisions in the version of that archived revision up to to the archived revision itself.

Thus, for example, Candice's tag revision has Alice and Bob's `patch-1` revision as an ancestor because it has logs for Alice and Bob's revisions:

```
base-0
patch-1
```

And Candices's `patch-2` revision, which merges in changes from Alice and Bob's `patch-2` and `patch-3` , has both of those additional revisions as ancestors (see Updating from a Branched-from Version).

Automated ChangeLogs

The command `tla changelog` generates a GNU-style `ChangeLog` file from a patch log:

```
  % cd ~/wd

  % tla get -A candice@candice.net--2003-candice \
              hello-world--candice--0.1 \
              hw-C-latest
  [....]

  % cd ~/wd/hw-C-latest

  % tla changelog
  # do not edit -- automatically generated by arch changelog
  # arch-tag: automatic-ChangeLog-- [...]
  #

  2003-01-30 GMT  Tom (testing) Lord <lord@emf.net>       patch-2

      Summary:
        merge from mainline sources
      Revision:
        hello-world--candice--0.1--patch-2

      Patches applied:

        * lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-3
           added copywrong statements

        * lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-2
           commented return from main


      new files:
```

```
      {arch}/ [...] /hello-world--mainline--0.1 [...] /patch-2
      {arch}/ [...] /hello-world--mainline--0.1 [...] /patch-3

     modified files:
      hw.c main.c

     new patches:
      lord@emf.net--2003-example/hello-world--mainline--0.1--patch-2
      lord@emf.net--2003-example/hello-world--mainline--0.1--patch-3


 2003-01-30 GMT  Tom (testing) Lord <lord@emf.net>       patch-1

     Summary:
       Punctuated the output correctly
     Revision:
       hello-world--candice--0.1--patch-1


     This program should say "hello, world" not "hello world".


     modified files:
      hw.c


 2003-01-30 GMT  Tom (testing) Lord <lord@emf.net>       base-0

     Summary:
       tag of lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-1
     Revision:
       hello-world--candice--0.1--base-0

     (automatically generated log message)


     new patches:
      lord@emf.net--2003-example/hello-world--mainline--0.1--base-0
      lord@emf.net--2003-example/hello-world--mainline--0.1--patch-1
```

Note that the generated `ChangeLog` includes a `tagline` . If you save the output of the `changelog` command in a project tree, either using tagline ids or giving it an explicit id that matches the `tagline` s id, the commands such as `commit` will automatically keep the `ChangeLog` up to date.

Development Branches -- The star-merge Style of Cooperation

up: arch Meets hello-world
next: Symbolic Tags
prev: Patch Logs and Project Tree History

In earlier chapters, we developed an extended example out of the `hello-world` project.

Alice and Bob, the primary programmers on the project, started one archive and created some revisions there.

Candice, a user of the project, created her own archive, started a branch of the `hello-world` project, and began maintaining her own local modifications.

In this chapter, we'll begin to consider a situation that is more typical of free software projects in the real world. Here, we'll consider Alice and Bob to be the maintainers of a public project, and Candice as a major remote contributor to the project. We'll identify the new revision control needs that arise from that arrangement, and look at some `arch` commands that help to satisfy those needs.

Promoting an Elementary Branch to a Development Branch

So far, if you've been following the examples, Candice has an elementary branch. She made a branch from the mainline, made some local changes, and has kept her branch up-to-date with Alice and Bob's mainline.

We're supposing, at this point, that Alice and Bob want to merge Candice's changes into the mainline.

Well, that merging work has already been done. Candice's latest revision is exactly the tree that Alice and Bob want. They can incorporate that merge into their mainline very simply, by committing Candice's latest revision to their own mainline:

```
% tla get -A candice@candice.net--2003-candice \
            hello-world--candice--0.1 \
            hw-C
[...]


% cd hw-C
```

```
        % tla set-tree-version -A lord@emf.net--2003-example \
                    hello-world--mainline--0.1

        % tla make-log
        ++log.hello-world--mainline--0.1--lord@emf.net--2003-example

        [... edit log file (consider `tla log-for-merge') ... ]

        % cat ++log.hello-world--mainline--0.1--lord@emf.net--2003-
example
        Summary: merge from Candice's Branch
        Keywords:

        Patches applied:

          * candice@candice.net--2003-candice/hello-world--candice--
0.1--patch-2
             merge from mainline sources

          * candice@candice.net--2003-candice/hello-world--candice--
0.1--patch-1
             Punctuated the output correctly

          * candice@candice.net--2003-candice/hello-world--candice--
0.1--base-0
             tag of
              lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-1

        % tla commit
        [....]
```

Read Carefully Note: Note carefully the *trick* we just used. Candice's latest revision was exactly what Alice and Bob wanted -- they combined `get` with `set-tree-version` to turn Candice's tree into one they could easily commit to their own mainline.

Simple Development Branches

Let's consider what happens as development proceeds on both branches. For this purpose, we'll introduce something new: a way of diagraming branches and the merges between them.

After the examples so far, we have this situation:

```
      mainline--0.1                    candice--0.1
      -------------                    ------------
        base-0             -----------> base-0 (a tag)
        patch-1  ---------'             patch-1
        patch-2             ----------> patch-2
        patch-3  ----------'  --------'
        patch-4  <-----------'
```

which tells us that the `candice` branch is a tag of `patch-1` from the mainline; that at `patch-2` of the `candice` branch, there was a merge of everything up to `patch-3` of the `mainline` ; and finally that `patch-4` of the mainline merges in everything up to `patch-2` from the `candice` branch.

Whenever we have a such a diagram in which none of the merge lines cross, that is a *simple development branch* .

The significance of a simple development branch is that it's a model for how two development efforts can work asynchronously on one project. Within each effort -- on each branch -- programmer's use the "update/commit" style of cooperation (see The update/commit Style of Cooperation). However, changes on one branch have no effect on the other until the two branches are merged.

Introducing The Development Branch Merging Problem

Let's suppose that more work happens on both the `mainline` and `candice` branches, leaving us with:

```
      mainline--0.1                    candice--0.1
      -------------                    ------------
        base-0             -----------> base-0 (a tag)
        patch-1  ---------'             patch-1
        patch-2             ----------> patch-2
        patch-3  ----------'  --------' patch-3
        patch-4  <-----------'          patch-4
        patch-5
        patch-6
        % tla revisions --summary -A candice@candice.net--2003-
candice \
                          hello-world--candice--0.1
        base-0
            tag of
            lord@emf.net--2003-example/hello-world--mainline--0.1--
patch-1
        patch-1
            Punctuated the output correctly
        patch-2
            merge from mainline sources
        patch-3
            added a period to output string
        patch-4
            capitalized the output string
```

```
        % tla revisions --summary -A lord@emf.net--2003-example \
                          hello-world--mainline--0.1
        base-0
            initial import
        patch-1
            Fix bugs in the "hello world" string
        patch-2
            commented return from main
        patch-3
            added copywrong statements
        patch-4
            merge from Candice's Branch
        patch-5
            fixed the copyrwrong for hw.c
        patch-6
            fixed the copyrwrong for main.c
```

Let's consider a scenario in which our goal is to merge the new work on the `mainline` branch into the `candice` branch. In other words, we want to wind up with:

```
mainline--0.1                    candice--0.1
-------------                    ------------
  base-0             -----------> base-0 (a tag)
  patch-1  ---------'             patch-1
  patch-2             ----------> patch-2
  patch-3  ----------'  --------' patch-3
  patch-4  <-----------'          patch-4
  patch-5               --------> patch-5
  patch-6  ------------'
```

How can we perform that merge? Let's start with the latest pre-merge candice revision (`patch-4`):

```
% tla get -A candice@candice.net--2003-candice \
               hello-world--candice--0.1--patch-4 \
               hw-C-4
[....]

% cd hw-C-4
```

Here are two techniques that **don't** work:

replay Does Not Solve the Development Branch Merge Problem

`replay` will try to apply all "missing" changes from the `mainline` into the `candice` tree. The list of changeset it will apply is given by:

```
% tla missing --summary \
              -A candice@candice.net--2003-example \
              hello-world--mainline--0.1
patch-4
    merge from Candice's Branch
patch-5
    fixed the copyrwrong for hw.c
patch-6
    fixed the copyrwrong for main.c
```

Problematic in that list is `patch-4` . It's a merge that includes all of the changes from the `candice` branch up to its `patch-2` level. Yet those changes are already present in the `patch-4` revision of the `candice` branch -- so `replay` will be applying them redundantly (cause patch conflicts).

Note of Warning: The `replay` command will not prevent you from running further replays even though the source tree is not in a consistant state. TLA in its current incarnation does not merge reject files. This leaves open the possibility that patch rejects will be lost if a second `replay` is performed before the rejects from the first replay are resolved. (Some day TLA may be able to merge multiple rejects into a combined reject.)

Advanced User Note: The `replay` command has options that would allows us to skip the `patch-4` revision from the mainline. That *sort of* solves the problem, but it has some drawbacks. First, it means that `patch-4` will continue to appear in the `missing` output of the `candice` branch. Second, there is nothing that guarantees us that the `patch-4` changeset contains *only* merges from the `candice` branch. If Alice and Bob made other changes in `patch-4` , and we skip that changeset, those other changes will be lost.

update Does Not Solve the Development Branch Merge Problem

Suppose we try to `update` from the `mainline` branch. Recall that `update` will compute a changeset from the youngest `mainline` ancestor of the project tree to the tree itself, then apply that changeset to the latest `mainline` revision.

We have a notation for this. A changeset from `X` to `Y` is written:

```
        delta(X, Y)
```

In this case, `update` will start by computing a changeset from the `mainline` `patch-3` revision to our project tree:

```
        delta(mainline--0.1--patch-3, hw-C-4)
```

The tree that results for applying a changeset from `X` to `Y` to a tree `Z` is written:

```
        delta(X, Y) [ Z ]
```

In other words, the result of `update` in our example can be described as:

```
        delta(mainline--0.1--patch-3, hw-C-4) [mainline--0.1--patch-6]
```

Here's the problem, though. The `patch-3` revision of `mainline` was not previously merged with the `candice` branch. Thus, the changeset

```
        delta(mainline--0.1--patch-3, hw-C-4)
```

will include, among other changes, the changes from `patch-1` and `patch-2` of the candice branch.

Unfortunately, the tree we'll be applying that changeset to, `mainline--0.1--patch-6` , has already been merged with `base-0...patch-2` of the `candice` branch.

As with `replay` , `update` will cause merge conflicts by making zredundant changes.

Solving One Instance of the Development Branch Merging Problem

Using just our `delta` notation and merge diagrams, let's look at solving this merge problem cleanly.

Remember that we currently have:

```
      mainline--0.1                     candice--0.1
      -------------                     ------------
        base-0             -----------> base-0 (a tag)
        patch-1  ---------'             patch-1
        patch-2             ----------> patch-2
        patch-3  ----------'  --------' patch-3
        patch-4  <-----------'          patch-4
        patch-5
        patch-6
```

and our goal is to create a new merge, for patch-5 of Candice's branch:

```
                              --------> patch-5
        patch-6  ------------'
```

We might decide to start with a `mainline` branch and merge in missing `candice` changes, or start with a `candice` tree and merge in missing `mainline` changes. Let's assume the latter (merging into a `candice` tree).

In this case, `mainline-0.1` revision `patch-6` is "up to date" with `candice-0.1` revision `patch-2`. We want too apply all changes since then to the latest `candice` revision:

```
        with:
                ancestor := candice--0.1--patch-2
                merge_in := mainline--0.1--patch-6
                target   := canidice--0.1--patch-4


        answer := delta(ancestor, merge_in)[target]
```

The arrows in the merge diagram are critical to figuring out the right answer. For example, suppose that the arrow from Candice's `patch-2` to the `mainline` revision `patch-4` wasn't there. Then the answer would be:

```
        with:
                ancestor := mainline--0.1--patch-3
                merge_in := mainline--0.1--patch-6
                target   := canidice--0.1--patch-4


        answer := delta(ancestor, merge_in)[target]
```

Tracing out the arrows for a given merge is a tedious process. It's automated by the `star-merge` command:

star-merge -- Solving the Development Branch Merging Problem in General

It's a bit beyond the scope of this tutorial to explain the complete solution to the development branch merging problem in general. The two solutions shown above illustrate two cases, but slightly different solutions are sometimes necessary.

What you should know is that when you have simple development branches (see Simple Development Branches), the command `star-merge` knows how to merge between them without causing spurious merge conflicts.

In ordinary use, you invoke `star-merge` in the tree you want to merge info, providing as an argument the tree you want to merge from:

```
% tla get -A candice@candice.net--2003-candice \
        hello-world--candice--0.1--patch-4 \
        merge-temp

% tla star-merge lord@emf.net--2003/hello-world--mainline--0.1
```

Symbolic Tags

up: arch Meets hello-world
next: Cherrypicking Changes
prev: Development Branches -- The star-merge Style of Cooperation

As projects grow larger and more complicated, it is often useful to be able to give a symbolic name to particular revisions within an arch version.

For example, let's suppose that the `hello-world` project has many revisions:

```
        mainline
        --------
        base-0
        patch-1
        patch-2
        ....
        patch-23
```

It may be that, as development proceeds, occasional "snapshot" releases are made from the `mainline` . Not every revision becomes a snapshot, but some do.

It would be convenient to provide a label of which revisions became snapshots:

```
        mainline
        --------
        base-0
        patch-1         snapshot 0
        patch-2
        ....
        patch-12        snapshot 2
        ....
        patch-23        snapshot 3
```

The `tag` command, introduced earlier, can be used for this purpose (see Making a Branch from a Remote Project in a Local Archive).

When we first encountered `tag` , it was used just to create the `base-0` revision of an elementary branch. It can also be used to create a branch *all* of whose revisions are tags.

Let's suppose that we'll be creating a branch called `hello-world--snapshots--0.1` . Diagramatically, we'll have:

```
mainline                        snapshots
--------                        ---------
base-0                --------> base-0 (tag)
patch-1 -------------'  ------> patch-1 (tag)
patch-2                '
....                  '
patch-12 ------------'
....
patch-23
```

To create the `snapshot` tag for `patch-23` :

```
% tla tag hello-world--mainline--0.1--patch-23 \
            hello-world--snapshots--0.1
```

after which we'll have:

```
mainline                        snapshots
--------                        ---------
base-0                --------> base-0 (tag)
patch-1 -------------'  ------> patch-1 (tag)
patch-2                ' -----> patch-2 (tag)
....                  ' '
patch-12 ------------' '
```

```
        ....                  '
        patch-23 -----------'
```

In effect, the `snapshots` branch is a kind of "symbolic name" with history. We can get the latest revision named by that symbol with:

```
        % tla get hello-world--snapshots--0.1
```

and earlier revisions by naming specific revisions, e.g.:

```
        % tla get hello-world--snapshots--0.1--patch-1
```

Usage Caution: As a rule of thumb, your branches should be either `commit` based branches (all revisions after `base-0` are created by `commit`) or tag-based branches (all revisions are created by `tag`). Commands such as `replay` , `update` , and `star-merge` are based on the presumption that you stick to that rule. While it can be tempting, in obscure circumstances, to mix `commit` and `tag` on a single branch -- it isn't generally recommended.

Cherrypicking Changes

up: arch Meets hello-world
next: Multi-tree Projects and Configuration Management
prev: Symbolic Tags

So far we've learned about elementary branches for maintaining changes apart from a primary development branch and development branches for coordinating asynchronous work on a single project (see Elementary Branches -- Maintaining Private Changes and Development Branches -- The star-merge Style of Cooperation).

In this chapter, we'll briefly describe a third kind of branch that's useful when a project consists of multiple "forks" -- multiple, equally primary branches.

Let's suppose, somewhat abstractly, that Alice and Bob's mainline has grown quite large:

```
mainline
--------
base-0
patch-1
....
patch-23
patch-24
patch-25
...
patch-42
```

At some point, perhaps because some controversy has emerged over choices made in the `mainline`, a new developer, Derick, declares a fork and starts his own branch:

```
mainline                derick
--------                ------
base-0          ------> base-0
patch-1        '
```

```
....          '
patch-23 ----'
patch-24
patch-25
...
patch-42
```

We already know that Derick can use `update` or `replay` to keep current with the mainline, but what he doesn't want to? What if Derick wants the changes in `patch-25` and `patch-42` , but none of the other post-`patch-23` changes from the `mainline` ?

Derick can apply specific changes from the `mainline` by specifying the exact revision he wants, rather than just specifying a version:

```
% cd ~/wd

% tla get hello-world--derick--0.1 derick

% cd derick

% tla replay -A lord@emf.net--2003-example \
         hello-world--mainline--0.1--patch-23

% tla replay -A lord@emf.net--2003-example \
         hello-world--mainline--0.1--patch-42

% tla missing -A lord@emf.net--2003-example \
         hello-world--mainline--0.1
patch-24
patch-25
...
patch-41


% tla logs -A lord@emf.net--2003-example \
         hello-world--mainline--0.1
base-0
patch-1
...
```

```
        patch-22
        patch-23
        patch-42
```

Cherrypicking changes in this manner isn't necessarily easy or even practical. It depends, for example, on the `mainline` changes being "clean changesets" (see Using commit Well -- The Idea of a Clean Changeset).

Nevertheless, for some projects, especially those characterized by lots of "forks", this technique can be useful.

Learning Note: Multiple revisions may be replayed with a single command, simply by giving all of them on the command line at once. The `replay` command also has a `--list` option which can useful for cherrypicking many changes at once. If you find yourself replaying specific revisions often, you should take a look at the `--list` option in `tla replay --help`.

Multi-tree Projects and Configuration Management

up: arch Meets hello-world
next: Revision Library Basics
prev: Cherrypicking Changes

You can define *meta-projects* which are combinations of individual projects that are separately tracked by `arch` . This allows you to divide a large project into smaller, more manageable pieces, each of which can develop independently of the others, and each of which can be a part of more than one meta-project.

This is accomplished by writing *config specs* , which define the contents of the meta-project and how they should be arranged in a source tree.

For example, `arch` itself is a meta-project. The source tree contains:

```
        dists/
          dists/src/
            dists/src/arch/
            dists/src/file-utils/
            dists/src/ftp-utils/
            dists/src/hackerlab/
            dists/src/shell-utils/
```

Each of those directories is the root of a project tree (contains a subdirectory named *{arch}*).

The topmost directory, `dists` also contains a subdirectory named `configs` . In that subdirectory are the meta-project configuration files. For example:

```
        dists/
          dists/configs/
            dists/configs/regexps.com/  # Tom's configuration files
              dists/configs/regexps.com/devo.arch
              dists/configs/regexps.com/release-template.arch
```

Here are the contents of `devo.arch` :

```
     #
     # Check out an arch distribution from the devo branches.
     # Latest revisions.
     #



     ./src                    lord@regexps.com--2002/package-
framework--devo
     ./src/arch               lord@regexps.com--2002/arch--devo
     ./src/file-utils         lord@regexps.com--2002/file-utils--devo
     ./src/ftp-utils          lord@regexps.com--2002/ftp-utils--devo
     ./src/hackerlab          lord@regexps.com--2002/hackerlab--devo
     ./src/shell-utils        lord@regexps.com--2002/shell-utils--devo
     ./src/text-utils         lord@regexps.com--2002/text-utils--devo
```

Each (non-blank, non-comment) line in that file has the format:

```
        LOCATION                    CONTENTS
```

which means, to create the meta-project, get the revision indicated by `CONTENTS` and install it at `LOCATION` . The `CONTENTS` field can be a branch (meaning, get the latest revision of the latest version on that branch), a version (meaning get the latest revision in that version), or a revision name (meaning get that revision, exactly).

To check out an entire `arch` tree, I first check out `dists` from `devo` , then use `build-config` :

```
        % tla get dists--devo dists
        [....]



        % cd dists



        % tla build-config regexps.com/dists.devo
        [....]
```

Once you have a meta-project tree, some other useful commands are:

```
     cat-config : output information about a multi-project config
```

One use of that command is to generate a list of sub-projects to which some other command can be iteratively applied:

```
% tla cat-config CFGNAME | awk '{print $1}' | xargs ...
```

Additionally, the option `--snap` can be usefully applied to a configuration that names subproces by version rather than revision. It examines the project tree to see what revisions are actually installed at each of the `LOCATIONs` . Then it writes a new config which specify those `REVISIONS` precisely. This is useful, for example, for recording the specific revisions you are about to turn into a distribution.

Revision Library Basics

up: arch Meets hello-world
next: Advanced Revision Library Use
prev: Multi-tree Projects and Configuration Management

For many purposes, it is useful to have a library containing pristine trees of a large number of revisions -- for example, all of the revisions in a particular version. To be practical, though, such a library must be represented in a space-efficient way.

Unix hard-links provide a natural way to store such a library. Each successive revision in a series is a copy of the previous, but with unmodified files shared via hard-links.

`arch` provides commands to help you build, maintain, and browse such a library.

As a pleasant side effect, many `arch` commands are speeded up if the revisions they need to operate are present in your revision library. You can read more about this in the next chapter.

Your Revision Library Locations

To begin a new revision library, first create a new directory (`DIR`) and then register its location:

```
        % tla my-revision-library DIR
```

You can check the location of your library with:

```
        % tla my-revision-library
```

or unregister it with:

```
        % tla my-revision-library -d DIR
```

Note that you can have more than one revision library: in effect you have a "path" listing all of your library locations.

Revision Library Format

A revision library has subdirectories of the form:

```
        ARCHIVE-NAME/CATEGORY/BRANCH/VERSION/REVISION/
```

Each `REVISION` directory contains the complete source of a particular revision, along with some supplemantary subdirectories and files:

```
        REVISION/,,patch-set/

                The patch set that creates this revision from
                its ancestor (unless the revision is a full-source
                base revision).
```

Although the permissions on files in the revision library are determined as determined by patch sets, **you must never modify files int the revision library**. Doing so will cause odd errors and failures in various `arch` commands.

Adding a Revision to the Library By Hand

You can add a selected revision to your revision library with:

```
        % tla library-add REVISION
```

`library-add` will normally add not only `REVISION` to the library, but all directly preceeding revisions (recursively) which are from the version as REVISION.

If you want to add only REVISION and no others, use the `--sparse` option:

```
        % tla library-add --sparse REVISION
```

Finding a Revision in the Library

You can find a particular revision in the library with `library-find` :

```
% tla library-find REVISION
PATH-TO-REVSION
```

The output is an absolute path name to the library directory containing the revision. (Once again, you must not modify files in that directory.)

Removing a Revision from the Library

To remove a particular revision from the library, use:

```
% tla library-remove REVISION
```

Be aware of the following limitation in the current release: suppose that you add three successive revisions, `A` , `B` , and `C` . Then you remove `B` , then re-add `B` . Now there is a chance that the file sharing between `B` and `C` will be less than optimal, causing your library to be larger than it needs to be. (You can fix this by then removing and re-adding `C` .)

Listing Library Contents

The command `library-archives` lists all archives with records in the library:

```
% tla library-archives
ARCHIVE-NAME
ARCHIVE-NAME
...
```

Similarly, you can list categories, branches, versions, or revisions:

```
% tla library-categories [ARCHIVE]
% tla library-branches [ARCHIVE/CATEGORY]
% tla library-versions [ARCHIVE/BRANCH]
% tla library-revisions [ARCHIVE/VERSION]
```

Individual Files in the Revision Library

You can locate an individual file in a revision library with:

```
        % tla library-file FILE [REVISION]
        PATH
```

or obtain its contents with:

```
        % tla cat-library-file FILE [REVISION]
        ...file contents...
```

Both commands accept the options `--id` and `--this` . With `--id` , the argument `FILE` is interpreted as an inventory id, and the file with that id is found.

With `--this` , `FILE` is interpreted as a file relative to the current directory, which should be part of a project tree. The file's inventory id is computed and the corresponding file found in `REVISION` .

Determining Patch Set Prerequisits

```
          % tla touched-files-prereqs REVISION
```

That command looks at the patch set for `REVISION` and at all preceding patch sets in the same version (it searches your library rather than your repository for this purpose). It reports the list of patches that touch overlapping sets of files and directories -- in other words, it tells you what patches can be applied independently of others. The command has an option to exclude from consideration file names matching a certain pattern (e.g. *=README* or *ChangeLog*). It has an option to exclude from the output list patches which have already been applied to a given project tree. It has an option to report the specific files which are overlapped.

Advanced Revision Library Use

up: arch Meets hello-world
next: Driving Process Automation with arch Hooks
prev: Revision Library Basics

By default, when you `get` a revision from an archive, arch stores a "pristine copy" of that revision under the `{arch}` directory.

Also by default, when `get` a revison, arch builds the revision by searching for the `import` ancestor or the nearest archive-cached ancestor -- then applying later patches to construct the revision you want.

`get` and similar operations can be made both faster and more space efficient by using revision libraries. For example, if `get` finds the revision you asked for in a library, it will copy it directly from there (rather than building it by patching) and skip building a pristine copy under `{arch}` .

That's all well and good -- but it can be awkward to have to remember to `library-add` revisions to your library. This section will show how you can automate the process.

Greedy Revision Libraries

A *greedy revision library* has the property that whenever arch looks to see if the library contains a particular revision, if the library _doesn't_ contain that revision, arch will add it automatically.

You can make a particular revision library directory greedy with the command:

```
        % tla library-config --greedy DIR
```

Sparse Revision Libraries

When arch automatically adds a revision to a greedy library, normally it does it in the default manner of `library-add` : it adds previous revisions in the same version as well.

If you were adding a revision to a library by-hand you could avoid that behavior with the `--sparse` option to `library-add` . To obtain that behavior for automatically added revisions, use:

```
        % tla library-config --sparse DIR
```

which means that if a revision is automatically added to the library located at DIR, it is added as if the `--sparse` option to `library-add` were being used.

Hard Linked Project Trees

Warning: To save yourself some confusion, do not use the following feature unless you understand (a) what a hard-link is and (b) what it means for an editor to "break hard links when writing a file". If you understand those terms, and know that the editor you use does in fact break hard links, then feel free to use this feature.

You can very rapidly `get` a revision from a revision library not by copying it, but instead by making hard-links to it:

```
        % tla get --link REVISION
```

The `build-config` command has a similar option:

```
        % tla build-config --link REVISION
```

This can save considerable disk space and greatly speed up the `get` operation.

(There is, of course, a small chance that when you use a hard-linked tree something will go wrong and modify the files in the revision library. Arch will notice that if it happens and give you an error message advising you to delete and reconstruct the problematic revision in the library.)

Putting it All Together

To sum up, a very handy and efficient set up involves:

1) Create one or more revision library directories.

2) Make at least some of those libraries greedy and possibly sparse.

3) Use the `--link` option to `get` and `build-config` .

When you work this way, and arch needs to automatically add a revision to a library for you, it will search for a library on the appropriate device (for hard-links purposes). Among those it will search first for a library that already contains the same version as the revision you want and, failing that, for a greedy library.

Driving Process Automation with arch Hooks

up: arch Meets hello-world
next: Speeding up arch by Caching Revisions in Archives
prev: Advanced Revision Library Use

In some circumstances, it is very useful to trigger actions upon the detection of changes to an archive. For example, you might want to send an email notification whenever new revisions are checked in.

This process occurs through arch by use of hooks. Each time that arch performs a command that modifies an archive, arch will attempt to run ~/.arch-params/hook, which must be set as executable.

Aguments given to the hook $1 : action performed (e.g. *commit*)

Environment Variables

Arguments Given to hook

Whenever arch performs a command that affects an archive, arch will run hook with the first argument set as the action performed. If I user runs a command (such as make-archive) then hook will be called multiple times with multiple arguments (such as make-archive, make-category, make branch and make-version)

The arguments that may be seen are:

import, commit, tag, make-archive, make-category, make-branch and make-version.

Environment Variables Passed to hook

Tla also passes certain variables to the hook when appropriate. Variables passed by Tla are prefaced with ARCH_. Variables that may be passed include:

Name : ARCH_ARCHIVE Description : The archive involved in the action Seen : all actions Example : lord@emf.net--2003-example

Name : ARCH_CATEGORY Description : Name of category created Seen : make-category Example : hello-world

Name : ARCH_BRANCH Description : Name of branch being created Seen : make-branch Example : mainline

Name : ARCH_VERSION Description : Name of version being created Seen : make-version Example : `0`.1

Name : ARCH_REVISION Descriptoin : Name of revision involved Seen : import, tag, commit Example : patch-6

Name : ARCH_LOCATION Description : Location of archive being created Seen : make-archive Example : /usr/lord/`archives`/2003-example

Name : ARCH_TREE_ROOT Description : Seen : commit, import Example : /home/lord/wd

Name : ARCH_TAGGED_ARCHIVE Description : Seen : tag Example :

Name : ARCH_TAGGED_REVISION Description : Seen : Example :

An Example of Using hook

```
#!/bin/sh


if [ "$1" == "commit" ]; then
   tla push-mirror lord@emf.net--2003-example \
      lord@emf.net--2003-example-MIRROR;
   fi
```

A more complex Examples of Using hook

```
#!/bin/sh


case "$1" in
   commit)
      case "$ARCH_CATEGORY" in
         hello-world)
            case "$ARCH_BRANCH" in
              mainline)
                   RELEASETYPE="stable"
```

```
                    ;;
                    devel)
                       RELEASETYPE="unstable"
                    ;;
                    *)



                  echo "The $RELEASETYPE version of Hello, World been
upgraded. \
                     New versions are available at ftp.hello.com" |\
                     mailto hello-users@hello.com -s "Hello upgraded"
               ;;
               goodbye-world)
                  case "$ARCH_BRANCH" in
                    mainline)
                         RELEASETYPE="stable"
                    ;;
                    devel)
                       RELEASETYPE="unstable"
                    ;;
                       RELEASETYPE="[unknown]"
                    *)
                  esac;
                  echo "The stable version of Goodbye, Cruel World
been upgraded. \
                     New versions are available at ftp.hello.com" |\
                     mailto hello-users@hello.com -s "Hello upgraded"
               ;;
             esac
         ;;
     esac
```

Robustness Issues with hook

Unfortunately, some fundamental physical properties of the universe make it impossible for arch to guarantee that hook will be invoked only once for each new category, branch, version, or revision. A (presumably rare) well timed interrupt or system failure can cause `notify` to invoke actions more than once for a given change to the archive.

Consequently, actions should be designed to be robust against that eventuality.

Additionally, if arch has been run concurrantly, then the hook may run concurrantly as well. This means that projects using hook should take care that hook is capable of running with simultaneous copies.

Speeding up arch by Caching Revisions in Archives

up: arch Meets hello-world
next: The arch Changeset Format
prev: Driving Process Automation with arch Hooks

This chapter will teach you one technique for speeding up access to an `arch` archive.

Consider an `arch` version that contains many revisions:

```
        mainline
        --------
        base-0
        patch-1
        ....
        patch-23
        patch-24
        patch-25
        ...
        patch-42
```

Suppose that a user (with no local pristine cache) wants to `get` the `patch-42` revision. `get` proceeds by first getting and unpacking the `base-0` revision, then getting each `patch-<N>` changeset, in order, and applying those to the tree.

If the list of changesets that need to be applied is long, or the sum of their sizes large in comparison to the tree side, then this implementation of `get` is needlessly inefficient.

One way to speed up `get` is by *archive caching revisions* -- storing "pre-built" copies of some revisions with the archive.

For example, the command:

```
        % tla cacherev -A lord@emf.net--2003-example \
        hello-world--mainline--0.1--patch-40
```

will build the `patch-40` revision, package it up as a tar bundle, and store a copy of that tar bundle in the `patch-40` directory of the archive.

Subsequently, a `get` of `patch-42` will work by first fetching the cached copy of the `patch-40` revision, then getting and applying the changesets for `patch-41` and `patch-42` : a savings of `40` changesets.

Usage Note: At this time, it's left up to you to decide which revisions to cache and which not. You might decide, for example, to automatically cache certain revisions from a `cron` job or to simply cache revisions by-hand whenever you notice that `get` is too slow. In the future, we hope to add better support for automatically caching revisions.

The arch Changeset Format

up: arch Meets hello-world
next: Customizing the inventory Naming Conventions
prev: Speeding up arch by Caching Revisions in Archives

An arch changeset is a directory containing a number of files and subdirectories. Each is described below.

Files:

```
orig-dirs-index
mod-dirs-index
orig-files-index
mod-files-index
```

Format:

```
<file path><tab><id>
```

Sorting:

```
sort -k 2
```

These contain indexes for all files and directories added, removed, or modified between the two trees.

Files:

```
original-only-dir-metadata
modified-only-dir-metadata
```

Format:

```
<metadata><tab><name>
```

Sorting:

```
sort -t '<tab>' -k 2
```

The field `<metadata>` contains literal output from the program `file-metadata` given the options `--permissions` . Some example output is:

```
--permissions 777
```

That output is also suitable for use as options and option arguments to the program `set-file-metadata` . Future releases `arch` might add additional flags (beside just `permissions`).

The list records the file permissions for all directories present in only one of the two trees.

Directories:

```
removed-files-archive
new-files-archive
```

Each of these directories contains complete copies of all files that occur in only the original tree (`removed-files-archive`) or modified tree (`new-files-archive`). Each saved file is archived at the same relative location it had in its source tree, with permissions (at least) preserved.

Directory:

```
patches
```

This directory contains a tree whose directory structure is a subset of the directory structure of the modified tree. It contains modification data for directories and files common to both trees.

For a file stored in the modified tree at the path `new_name` , the `patches` directory may contain:

```
new_name.link-orig

        The original file is a symbolic link.
```

`new_name.link-orig' is a text file containing the target of that link plus a final newline.

This file is only present if link target has changed, or if the link was replaced by a regular file.

new_name.link-mod

The modified file is a symbolic link and this file is a text file containing the target for the link plus a final newline.

This file is only present if the link target has changed, or if the link replaces a regular file.

new_name.original

This is a complete copy of the file from the original tree, preserving (at least) permissions.

This file is only present if the file was replaced by a symbolic link, or if the file contents can not be handled by `diff(1)'.

new_name.modified

This is a complete copy of the file from the modified tree, preserving (at least) permissions.

This file is only present if the file replaces a symbolic link, or if the file contents can not be handled by `diff(1)'.

new_name.patch

This is a standard context diff between the original file and modified file. One popular version of diff (`GNU diff') generates non-standard context diffs by omitting one copy of lines of context that are identical between the original and modified file, so

```
                for now, `.patch' files may have the same bug.
                Fortunately, the only popular version of `patch'
                (``GNU patch'') is tolerant of receiving such input.



        new_name.meta-orig
        new_name.meta-mod

                File metadata (currently only permissions) changed
                between the two versions of the file.  These files
                contain output from the `file-metadata' program with
                the flags `--symlink --permissions', suitable for
                comparison to similar output, and for use as options
                and option arguments to `set-file-metadata'.

                These files are also included if a regular file has
                replaced a symbolic link or vice versa.



        new_name/=dir-meta-orig
        new_name/=dir-meta-mod

                Directory metadata (currently only permissions)
changed
                between the two versions of the directory containing
                these files.  These files contain output from the
                `file-metadata' program with the flags `--symlink
                --permissions', suitable for comparison to similar
                output, and for use as options and option arguments to
                `set-file-metadata'.
```

Note: If a regular file (or symbolic link) replaces a directory, or vice versa, this is recorded as a file (or link) removed (or added) in one tree and added (or removed) in the other.

Customizing the inventory Naming Conventions

up: arch Meets hello-world
next: The GNU General Public License
prev: The arch Changeset Format

In Project Tree Inventories, you learned how the `tla inventory` command classifies files within a project tree using a set of naming conventions. This appendix explains how you can customize those naming conventions.

When to Customize Naming Conventions

It's best to make customizations to the naming conventions of a project at the outset: before you `import` your first revision.

If you *must* make changes later, then it's **essential** that your changes do not change the classification of files already in the latest revision(s) of your project at the time you make the change (otherwise, you are likely to experience perplexing and undesirable behavior).

How to Customize Naming Conventions

You should begin by reviewing the naming convention algorithm in The arch Naming Conventions. You can modify that algorithm by changing the regular expression used for each category test.

You can customize naming conventions by modifying the file `./{arch}/=tagging-method` in your project trees. That file is created by the `id-tagging-method` command and initially, it contains a single line which names the id tagging method (`names` , `explicit` , `tagline` (or the now deprecated, but popular in some older projects, including arch itself, `implicit`)).

In particular, `=tagging-method` can contain blank lines and comments (lines beginning with `#`) and directives, one per line. The permissible directives are:

```
tagline
implicit
explicit
names
        specify the id tagging method to use for this tree
exclude RE
junk RE
backup RE
precious RE
unrecognized RE
source RE
        specify a regular expression to use for the indicated
        category of files.
```

Regular expressions are specified in Posix ERE syntax (the same syntax used by *egrep* , *grep -E* , and *awk*) and have default values which implement the naming conventions described in The arch Naming Conventions.

A given regexp directive can occur more than once, in which case the regexps are concatenated as alternatives. Thus, for example:

```
        source  .*\.c$
        source  .*\.h$
```

is equivalent to:

```
        source (.*\.c$)|(.*\.h$)
```

Per-Directory Regexps

A source directory can contain a `.arch-inventory` file.

`.arch-inventory` files can contain regexp declarations just like those in =tagging-method (i.e., one for `excludes` , one for `junk` , etc.) Let's call these the *dir-local regexps* . The `=tagging-method` regexps are the *global regexps* .

While traversing a tree, each file is classified-by-name as follows. the steps which are changed by `.arch-inventory` are marked with `[*]` :

```
        0) "." and ".." remain excluded files, no matter what.


   [*]  1) if excluded files are being omitted from the inventory,
           and either the dir-local or global regexp, the file
           is excluded


        2) if the file is a control file, it is source


        3) if the file falls into one of the "mandatory categories"
           (",," and "++" files) it is categorized as junk or
           precious respectively.


   [*]  4) the dir-local (only) regexps are tried in the usual order:
           junk, backup, precious, unrecognized, source.  If the file
           matches, it is suitably categorized.


        5) the global regexps are tried in the same order.


        6) otherwise the file is unrecognized.
```

The GNU General Public License

up: arch Meets hello-world
next: Uh....a Little Help Here?
prev: Customizing the inventory Naming Conventions

`arch` is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation (and reproduced below).

This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License (reproduced below) for more details.

I have chosen to use the GPL for this software because I believe it best reflects the duties to society of a software engineer. It is the best license for users, for my fellow engineers, and for society as a whole. As is beginning to be widely appreciated, this license is a startling profound and influential document and is worthy of study in its own right.

In a commercial climate that grew up mostly under proprietary licenses (those that fall far short of protecting the freedoms and promoting the obligations of the GPL), my choice of this license has, at the moment, made it difficult for me to recover the costs of developing arch and to make a profit from my work going forward. Those are very serious problems, in my opinion. Please see also Uh....a Little Help Here?.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the

following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so

long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO

WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
    <one line to give the program's name and a brief idea of what it
    does.>
    Copyright (C) <year> <name of author>

    This program is free software; you can redistribute it and/or
    modify it under the terms of the GNU General Public License as
    published by the Free Software Foundation; either version 2 of
```

> the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type
`show w'.  This is free software, and you are welcome to
redistribute it under certain conditions; type `show c' for
details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Uh....a Little Help Here?

up: arch Meets hello-world
next: Indexes
prev: The GNU General Public License

`arch` is a *Community Supported Free Software Project* -- I rely on the financial support of the community to be able to develop `arch` and the other free software projects that I work on.

If you are able to help out, even just a little, please do so. I'm able to accept contributions as `lord@emf.net` on Paypal. Arrangements can be made to accept contributions larger than a few 10s of dollars as a tax-deductible contribution to a non-profit organization (contact me if you would like to do this.)

Finally, if you represent a business or non-profit organization, I offer a *Release Subscription Service* -- a formally invoiced mechanism, suitable for corporate purchasing practices, for contributing and gaining recognition in my eyes as a true customer for my development work. (Again, please contact me if this is of interest to you.)

Indexes

up: arch Meets hello-world
prev: Uh....a Little Help Here?