Monotone

A distributed version control system

Graydon Hoare

Complete table of contents

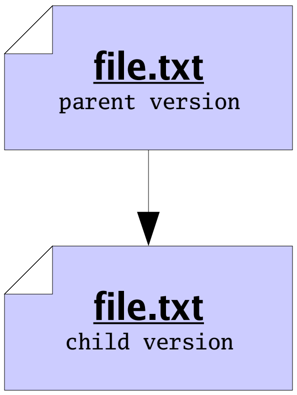
Table of Contents

1 Concepts

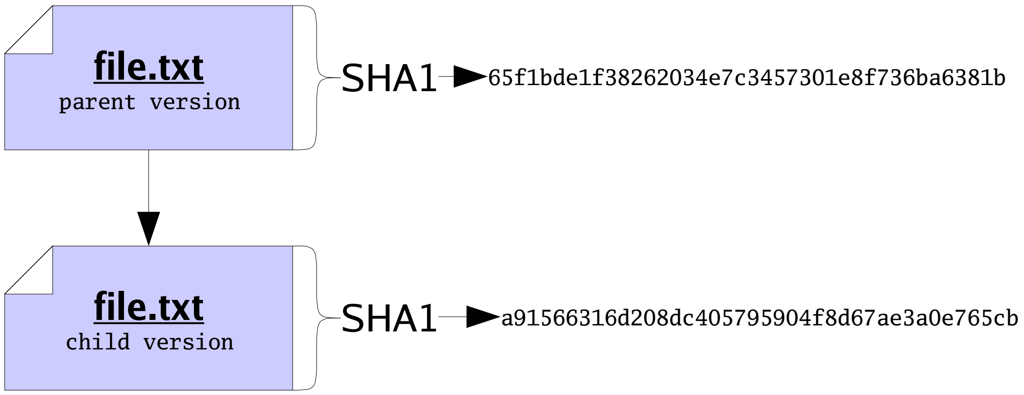
This chapter should familiarize you with the concepts, terminology, and behavior described in the remainder of the user manual. Please take a moment to read it, as later sections will assume familiarity with these terms.

1.1 Versions of files

Suppose you wish to modify a file '`file.txt`' on your computer. You begin with one *version* of the file, load it into an editor, make some changes, and save the file again. Doing so produces a new *version* of the file. We will say that the older version of the file was a *parent*, and the new version is a *child*, and that you have performed an *edit* between the parent and the child. We may draw the relationship between parent and child using a graph, where the arrow in the graph indicates the direction of the edit, from parent to child.



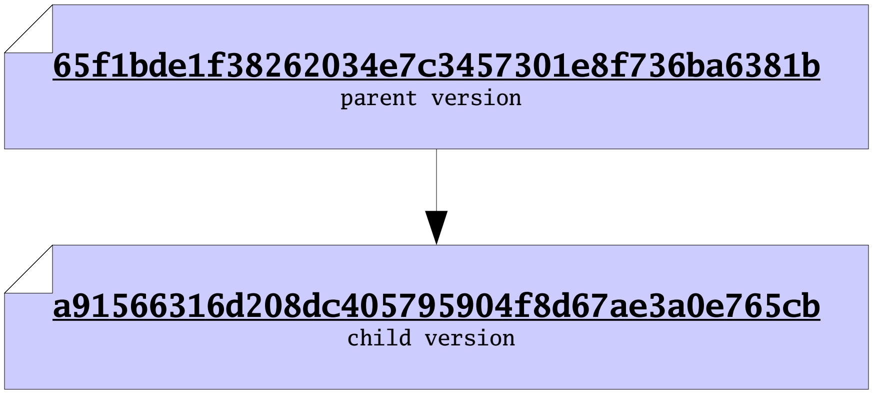
We may want to identify the parent and the child precisely, for sake of reference. To do so, we will compute a *cryptographic hash function*, called SHA1, of each version. The details of this function are beyond the scope of this document; in summary, the SHA1 function takes a version of a file and produces a short string of 20 bytes, which we will use to uniquely identify the version[1]. Now our graph does not refer to some "abstract" parent and child, but rather to the exact edit we performed between a specific parent and a specific child.


file.txt
parent version
SHA1
65f1bde1f38262034e7c3457301e8f736ba6381b
file.txt
child version
SHA1
a91566316d208dc405795904f8d67ae3a0e765cb

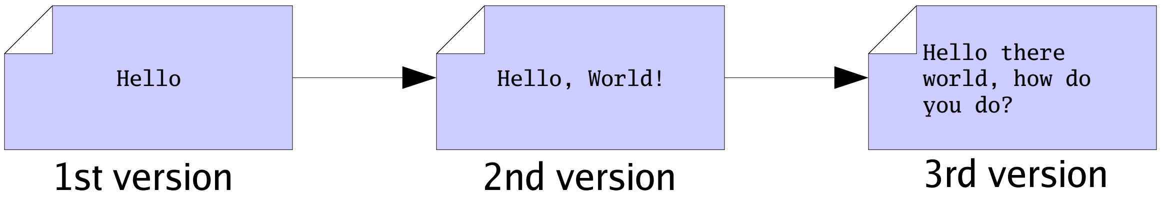
When dealing with versions of files, we will dispense with writing out "file names", and identify versions *purely* by their SHA1 value, which we will also refer to as their *file ID*. Using IDs alone will often help us accommodate the fact that people often wish to call files

[1] We say SHA1 values are "unique" here, when in fact there is a small probability of two different versions having the same SHA1 value. This probability is very small, so we discount it.

by different names. So now our graph of parent and child is just a relationship between two versions, only identified by ID.

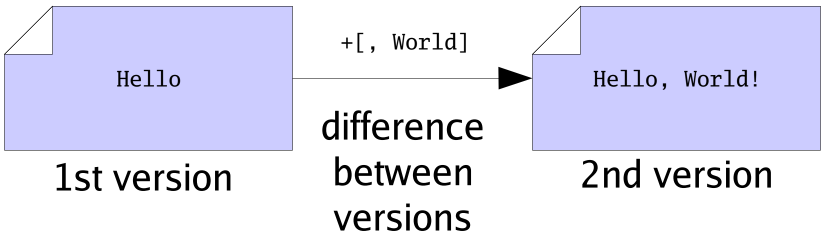

65f1bde1f38262034e7c3457301e8f736ba6381b
parent version
a91566316d208dc405795904f8d67ae3a0e765cb
child version

Version control systems, such as monotone, are principally concerned with the storage and management of *multiple* versions of some files. One way to store multiple versions of a file is, literally, to save a separate *complete* copy of the file, every time you make a change. When necessary, monotone will save complete copies of your files, compressed with the `zlib` compression format.



Often we find that successive versions of a file are very similar to one another, so storing multiple complete copies is a waste of space. In these cases, rather than store *complete* copies of each version of a file, we store a compact description of only the *changes* which are made between versions. Such a description of changes is called a *delta*.

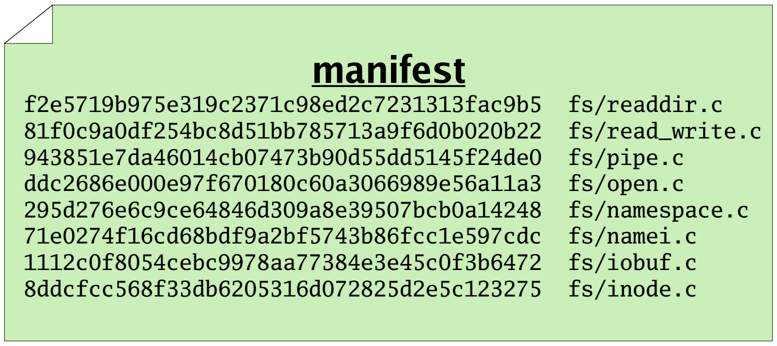
Storing deltas between files is, practically speaking, as good as storing complete versions of files. It lets you undo changes from a new version, by applying the delta backwards, and lets your friends change their old version of the file into the new version, by applying the delta forwards. Deltas are usually smaller than full files, so when possible monotone stores deltas, using a modified `xdelta` format. The details of this format are beyond the scope of this document.



1.2 Versions of trees

After you have made many different files, you may wish to capture a "snapshot" of the versions of all the files in a particular collection. Since files are typically collected into *trees* in a file system, we say that you want to capture a *version of your tree*. Doing so will permit you to undo changes to multiple files at once, or send your friend a *set* of changes to many files at once.

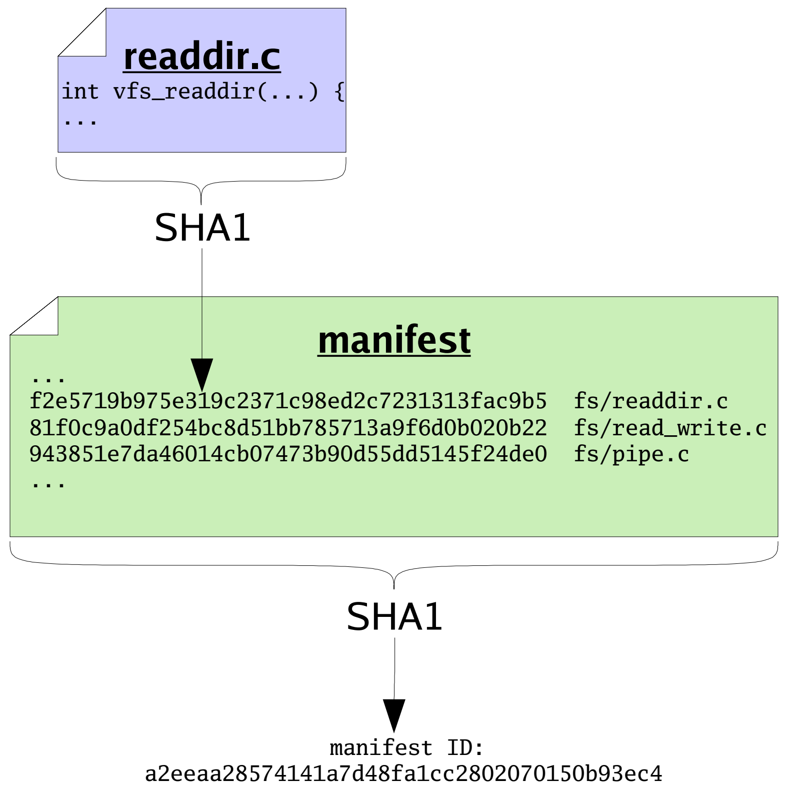
To make a snapshot of a tree, we begin by writing a special file called a *manifest*. In fact, monotone will write this file for us, but we could write it ourselves too. It is just a plain text file. Each line of a manifest file contains two columns. In the first column we write the ID of a file in your tree, and in the second column we write the path to the file, from the root of our tree to the filename.



manifest

```
f2e5719b975e319c2371c98ed2c7231313fac9b5  fs/readdir.c
81f0c9a0df254bc8d51bb785713a9f6d0b020b22  fs/read_write.c
943851e7da46014cb07473b90d55dd5145f24de0  fs/pipe.c
ddc2686e000e97f670180c60a3066989e56a11a3  fs/open.c
295d276e6c9ce64846d309a8e39507bcb0a14248  fs/namespace.c
71e0274f16cd68bdf9a2bf5743b86fcc1e597cdc  fs/namei.c
1112c0f8054cebc9978aa77384e3e45c0f3b6472  fs/iobuf.c
8ddcfcc568f33db6205316d072825d2e5c123275  fs/inode.c
```

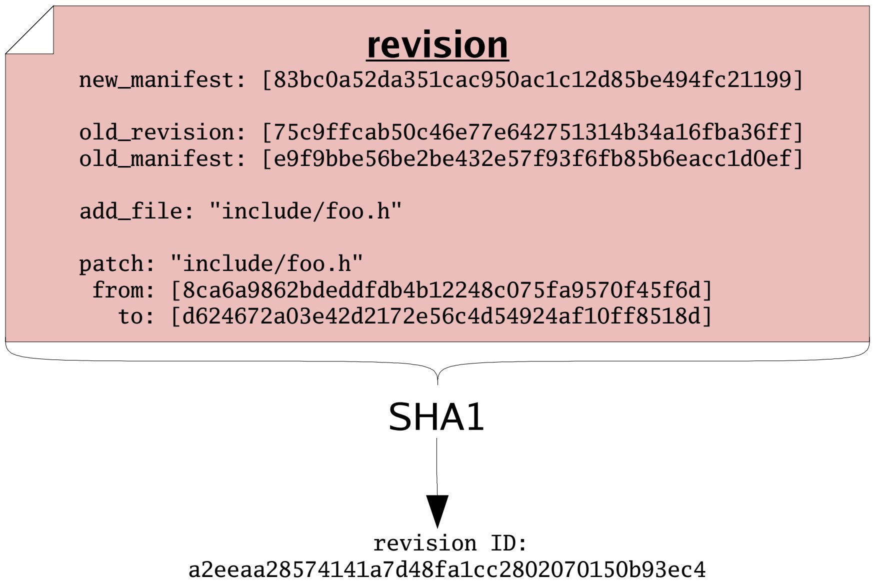
Now we note that a manifest is itself a file. Therefore a manifest can serve as input to the SHA1 function, and thus every manifest has an ID of its own. By calculating the SHA1 value of a manifest, we capture the *state of our tree* in a single *manifest ID*. In other words, the ID of the manifest essentially captures all the IDs and file names of every file in our tree, combined. So we may treat manifests and their IDs as *snapshots* of a tree of files, though lacking the actual contents of the files themselves.


readdir.c
int vfs_readdir(...) {
...
SHA1
manifest
...
f2e5719b975e319c2371c98ed2c7231313fac9b5 fs/readdir.c
81f0c9a0df254bc8d51bb785713a9f6d0b020b22 fs/read_write.c
943851e7da46014cb07473b90d55dd5145f24de0 fs/pipe.c
...
SHA1
manifest ID:
a2eeaa28574141a7d48fa1cc2802070150b93ec4

As with versions of files, we may decide to store manifests in their entirety, or else we may store only a compact description of changes which occur between different versions of manifests. As with files, when possible monotone stores compact descriptions of changes between manifests; when necessary it stores complete versions of manifests.

1.3 Historical records

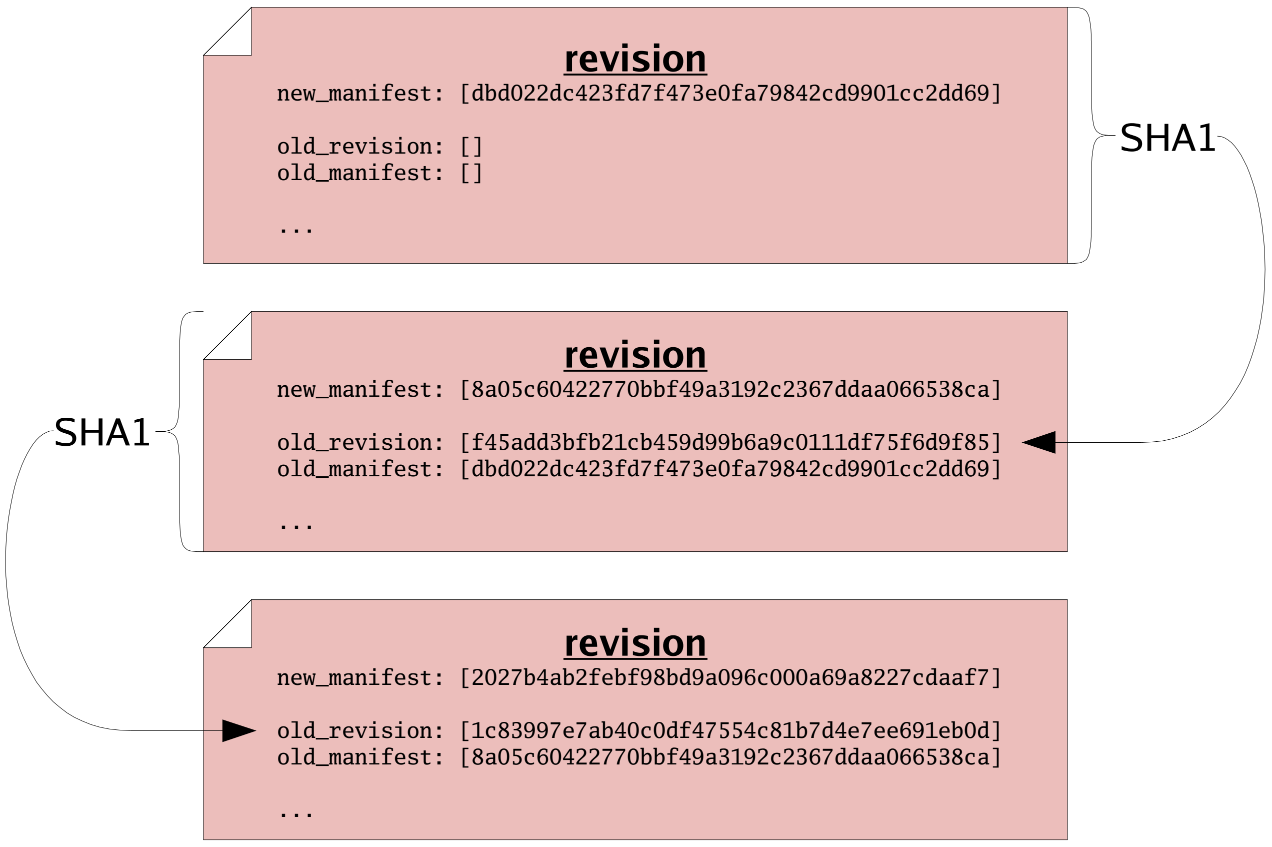
Suppose you sit down to edit some files. Before you start working, you may record a manifest of the files, for reference sake. When you finish working, you may record another manifest. These “before and after” snapshots of the tree of files you worked on can serve as historical records of the set of changes, or *changeset*, that you made. In order to capture a “complete” view of history – both the changes made and the state of your file tree on either side of those changes – monotone builds a special composite file called a *revision* each time you make changes. Like manifests, revisions are ordinary text files which can be passed through the SHA1 function and thus assigned a *revision ID*.



```
new_manifest: [83bc0a52da351cac950ac1c12d85be494fc21199]

old_revision: [75c9ffcab50c46e77e642751314b34a16fba36ff]
old_manifest: [e9f9bbe56be2be432e57f93f6fb85b6eacc1d0ef]

add_file: "include/foo.h"

patch: "include/foo.h"
 from: [8ca6a9862bdeddfdb4b12248c075fa9570f45f6d]
   to: [d624672a03e42d2172e56c4d54924af10ff8518d]
```

```
revision ID:
a2eeaa28574141a7d48fa1cc2802070150b93ec4
```

The content of a revision makes reference to file IDs, in describing a changeset, and manifest IDs, in describing tree states “before and after” the changeset. Crucially, revisions also make reference to *other revision IDs*. This fact – that revisions include the IDs of other revisions – causes the set of revisions to join together into a historical *chain of events*, somewhat like a “linked list”. Each revision in the chain has a unique ID, which includes *by reference* all the revisions preceeding it. Even if you undo a changeset, and return to a previously-visited manifest ID during the course of your edits, each revision will incorporate the ID of its predecessor, thus forming a new unique ID for each point in history.

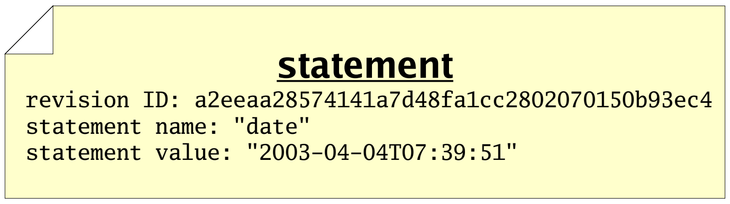

revision
new_manifest: [dbd022dc423fd7f473e0fa79842cd9901cc2dd69]
old_revision: []
old_manifest: []
...
SHA1
revision
new_manifest: [8a05c60422770bbf49a3192c2367ddaa066538ca]
old_revision: [f45add3bfb21cb459d99b6a9c0111df75f6d9f85]
old_manifest: [dbd022dc423fd7f473e0fa79842cd9901cc2dd69]
...
SHA1
revision
new_manifest: [2027b4ab2febf98bd9a096c000a69a8227cdaaf7]
old_revision: [1c83997e7ab40c0df47554c81b7d4e7ee691eb0d]
old_manifest: [8a05c60422770bbf49a3192c2367ddaa066538ca]
...

1.4 Certificates

Often, you will wish to make a *statement* about a revision, such as stating the reason that you made some changes, or stating the time at which you made the changes, or stating that the revision passes a test suite. Statements such as these can be thought of, generally, as a bundle of information with three parts:

- an *ID*, indicating which revision you are making a statement about
- a *name* indicating the type of statement you are making, such as "changelog", "date" or "testresult"
- a *value* indicating the remaining detail of the statement, such as "fixed a bug", "March 9th" or "1"

For example, if you want to say that a particular revision was composed on April 4, 2003, you might make a statement like this:



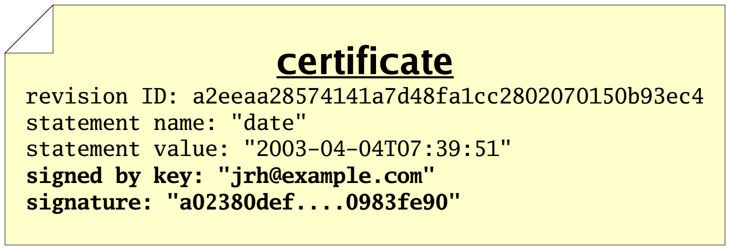
statement

```
revision ID: a2eeaa28574141a7d48fa1cc2802070150b93ec4
statement name: "date"
statement value: "2003-04-04T07:39:51"
```

In an ideal world, these are all the parts of a statement we would need in order to go about our work. In the real world, however, there are sometimes malicious people who would make false or misleading statements; so we need a way to verify that a particular person made a particular statement about a revision. We therefore will add two more pieces of information to our bundle:

- a *key* which identifies the person making a statement
- a *signature* — just a large number with particular properties — certifying the fact that the person made the statement

When these 2 items accompany a statement, we call the total bundle of 5 items a *certificate*, or *cert*. A cert makes a statement in a secure fashion. The security of the signature in a cert is derived from the RSA cryptography system, the details of which are beyond the scope of this document.



certificate

```
revision ID: a2eeaa28574141a7d48fa1cc2802070150b93ec4
statement name: "date"
statement value: "2003-04-04T07:39:51"
signed by key: "jrh@example.com"
signature: "a02380def....0983fe90"
```

Monotone uses certs extensively. Any "extra" information which needs to be stored, transmitted or retrieved — above and beyond files, manifests, and revisions — is kept in the form of certs. This includes change logs, time and date records, branch membership, authorship, test results, and more. When monotone makes a decision about storing, transmitting, or extracting files, manifests, or revisions, the decision is often based on certs it has seen, and the trustworthiness you assign to those certs.

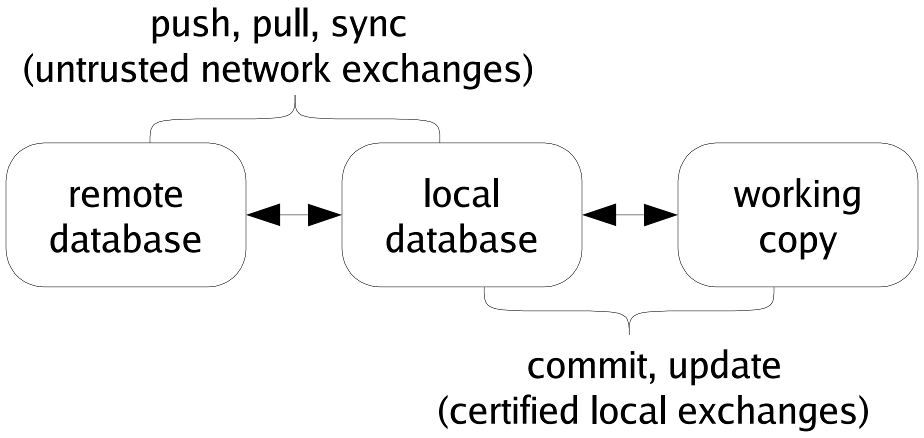
The RSA cryptography system — and therefore monotone itself — requires that you exchange special “public” numbers with your friends, before they will trust certificates signed by you. These numbers are called *public keys*. Giving someone your public key does not give them the power to *impersonate* you, only to verify signatures made by you. Exchanging public keys should be done over a trusted medium, in person, or via a trusted third party. Advanced secure key exchange techniques are beyond the scope of this document.

1.5 Storage and workflow

Monotone moves information in and out of three different types of storage:

- a *working copy* in the local file system
- a *local database* in the local file system
- a *remote database* elsewhere on the internet

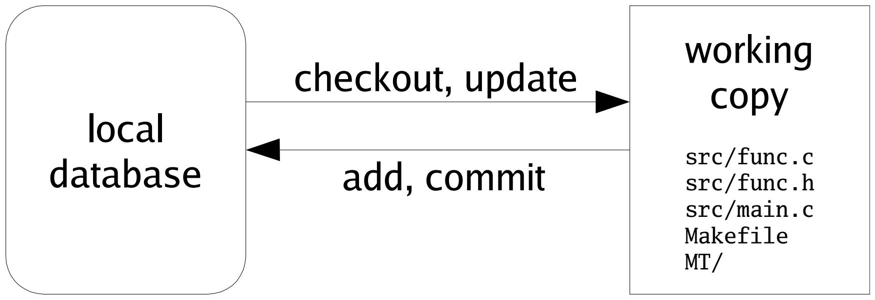
All information passes *through* your local database, en route to some other destination. For example, when changes are made in a working copy, you may save those changes to your database, and later you may synchronize your database with someone else's. Monotone will not move information directly between a working copy and a remote database, or between working copies. Your local database is always the "switching point" for communication.



A *working copy* is a tree of files in your file system, arranged according to the list of file paths and IDs in a particular manifest. A special directory called '`MT`' exists in the root of any working copy. Monotone keeps some special files in the '`MT`' directory, in order to track changes you make to your working copy.

Aside from the special '`MT`' directory, a working copy is just a normal tree of files. You can directly edit the files in a working copy using a plain text editor or other program; monotone will automatically notice when you make any changes. If you wish to add files, remove files, or move files within your working copy, you must tell monotone explicitly what you are doing, as these actions cannot be deduced.

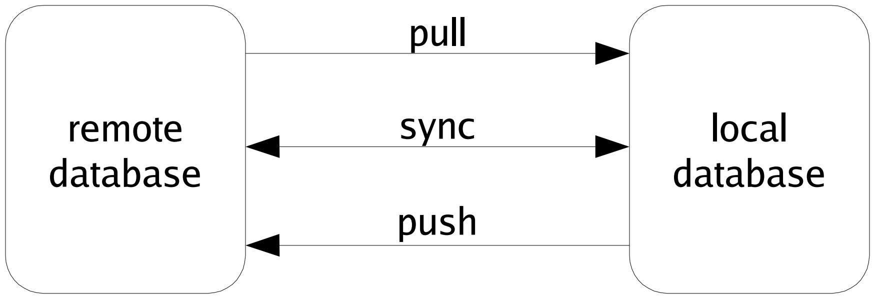
If you do not yet have a working copy, you can *check out* a working copy from a database, or construct one from scratch and *add* it into a database. As you work, you will occasionally *commit* changes you have made in a working copy to a database, and *update* a working copy to receive changes that have arrived in a database. Committing and updating take place purely between a database and a working copy; the network is not involved.



A *database* is a single, regular file. You can copy or back it up using standard methods. Typically you keep a database in your home directory. Databases are portable between different machine types. You can have multiple databases and divide your work between them, or keep everything in a single database if you prefer. You can dump portions of your database out as text, and read them back into other databases, or send them to your friends. Underneath, databases are accessed using a standard, robust data manager, which makes using even very large databases efficient. In dire emergencies, you can directly examine and manipulate a database using a simple SQL interface.

A database contains many files, manifests, revisions, and certificates, some of which are not immediately of interest, some of which may be unwanted or even false. It is a collection of information received from network servers, working copies, and other databases. You can inspect and modify your databases without affecting your working copies, and vice-versa.

Monotone knows how to exchange information in your database with other remote databases, using an interactive protocol called *netsync*. It supports three modes of exchange: pushing, pulling, and synchronizing. A *pull* operation copies data from a remote database to your local database. A *push* operation copies data from your local database to a remote database. A *sync* operation copies data both directions. In each case, only the data missing from the destination is copied. The netsync protocol calculates the data to send "on the fly" by exchanging partial hash values of each database.



In general, work flow with monotone involves 3 distinct stages:

- When you *commit* changes from your working copy to your database, your database stores the changes but does not communicate with the network. Your commits happen immediately, without consulting any other party, and do not require network connectivity.

- When you are ready to *exchange* work with someone else, you can push, pull, or sync with other databases on the network. When you talk to other servers on the network, your database may change, but your working copy will not. In fact, you do not need a working copy at all when exchanging work.
- When you *update* your working copy, some (but not all) of the changes which your database received from the network are applied to your working copy. The network is not consulted during updates.

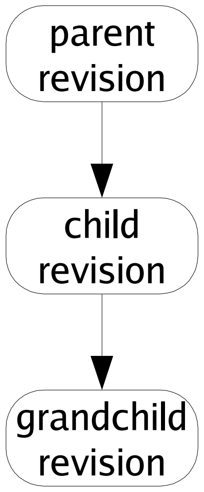
The last stage of workflow is worth clarifying: monotone does *not* blindly apply all changes it receives from a remote database to your working copy. Doing so would be very dangerous, because remote databases are not always trustworthy systems. Rather, monotone evaluates the certificates it has received along with the changes, and decides which particular changes are safe and desirable to apply to your working copy.

You can always adjust the criteria monotone uses to judge the trustworthiness and desirability of changes in your database. But keep in mind that it always uses *some* criteria; receiving changes from a remote server is a *different* activity than applying changes to a working copy. Sometimes you may receive changes which monotone judges to be untrusted or bad; such changes may stay in your database but will *not* be applied to your working copy.

Remote databases, in other words, are just untrusted “buckets” of data, which you can trade with promiscuously. There is no trust implied in communication.

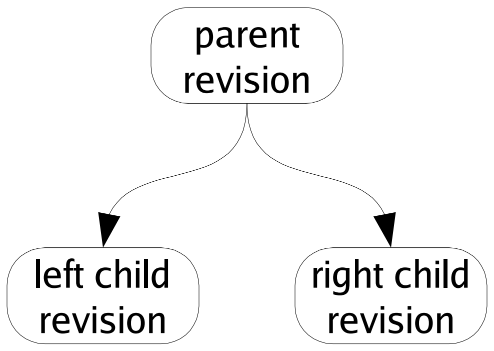
1.6 Forks and merges

So far we have been talking about revisions as though each logically follows exactly one revision before it, in a simple sequence of revisions.

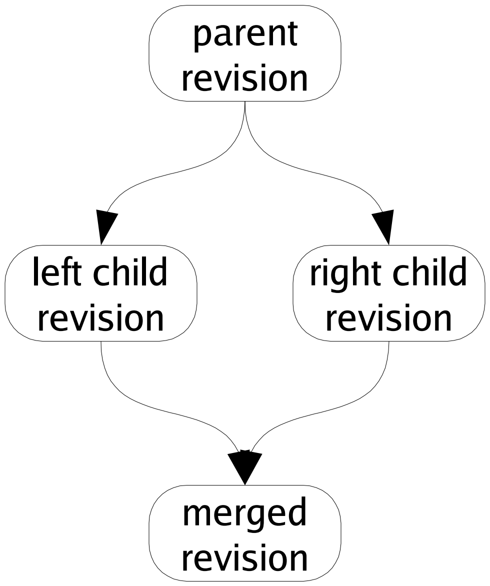


This is a rosy picture, but sometimes it does not work out this way. Sometimes when you make new revisions, other people are *simultaneously* making new revisions as well, and their revisions might be derived from the same parent as yours, or contain different changesets. Without loss of generality, we will assume simultaneous edits only happen two-at-a-time; in fact many more edits may happen at once but our reasoning will be the same.

We call this situation of simultaneous edits a *fork*, and will refer to the two children of a fork as the *left child* and *right child*. In a large collection of revisions with many people editing files, especially on many different computers spread all around the world, forks are a common occurrence.



If we analyze the changes in each child revision, we will often find that the changeset between the parent and the left child are unrelated to the changeset between the parent and the right child. When this happens, we can usually *merge* the fork, producing a common grandchild revision which contains both changesets.



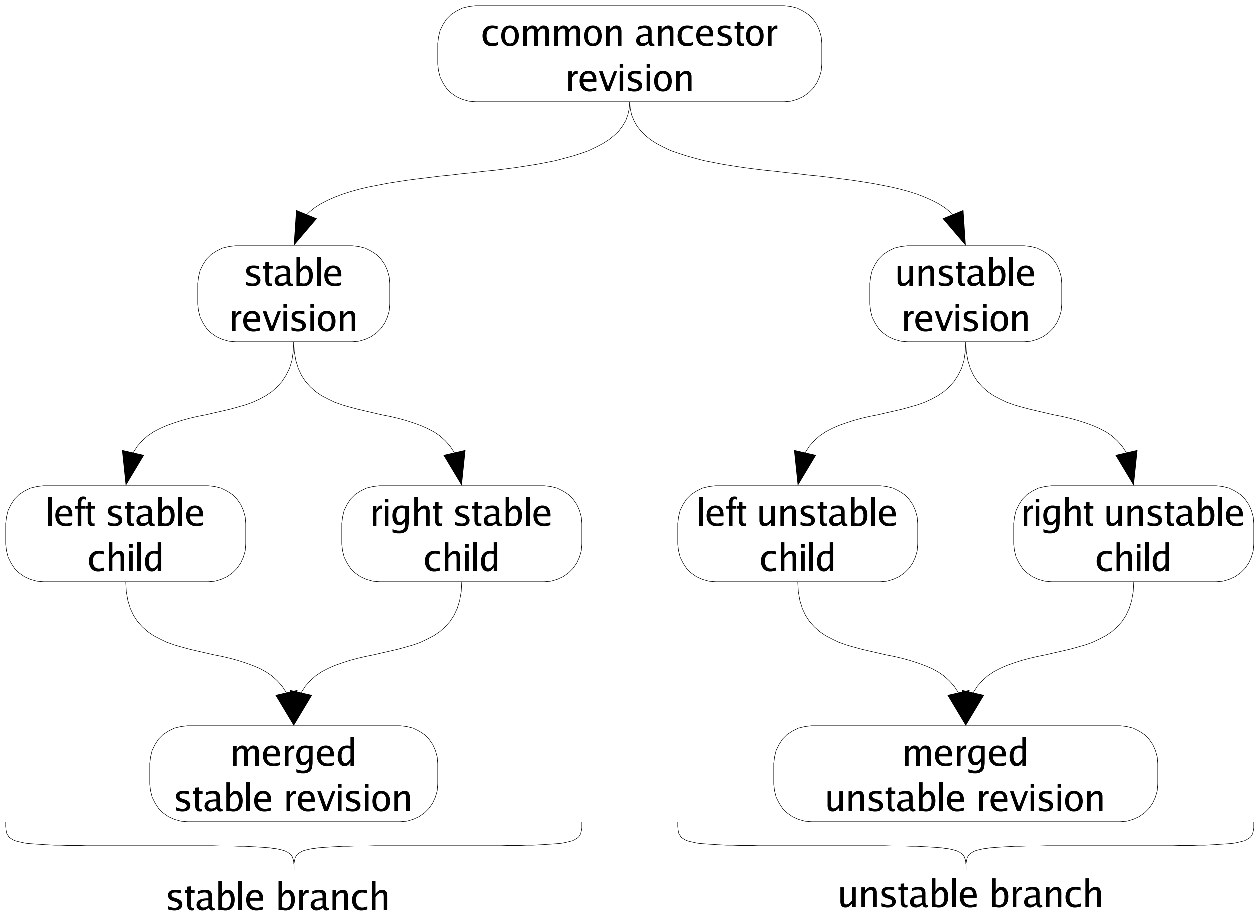
1.7 Branches

Sometimes, people intentionally produce forks which are *not supposed to be merged*; perhaps they have agreed to work independently for a time, or wish to change their files in ways which are not logically compatible with each other. When someone produces a fork which is supposed to last for a while (or perhaps permanently) we say that the fork has produced a new *branch*. Branches tell monotone which revisions you would like to merge, and which you would like to keep separate.

You can see all the available branches using `monotone list branches`.

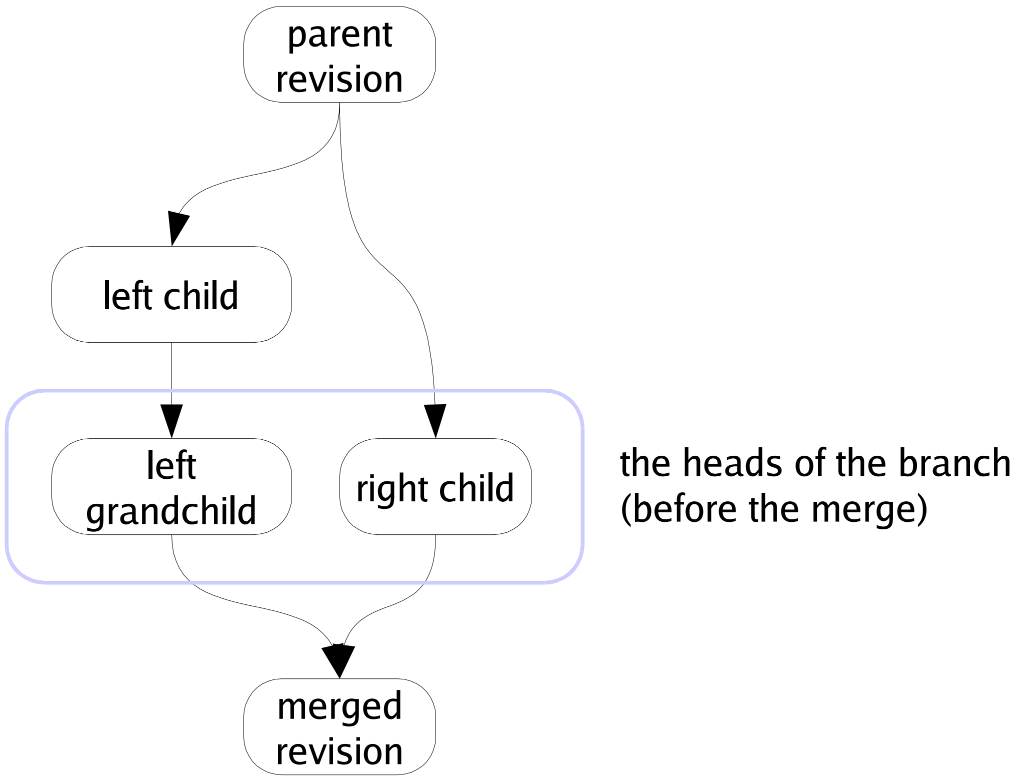
Branches are indicated with certs. The cert name `branch` is reserved for use by monotone, for the purpose of identifying the revisions which are members of a branch. A `branch` cert has a symbolic "branch name" as its value. When we refer to "a branch", we mean all revisions with a common branch name in their `branch` certs.

For example, suppose you are working on a program called "wobbler". You might develop many revisions of wobbler and then decide to split your revisions into a "stable branch" and an "unstable branch", to help organize your work. In this case, you might call the new branches "wobbler-stable" and "wobbler-unstable". From then on, all revisions in the stable branch would get a cert with name `branch` and value `wobbler-stable`; all revisions in the unstable branch would get a cert with name `branch` and value `wobbler-unstable`. When a `wobbler-stable` revision forks, the children of the fork will be merged. When a `wobbler-unstable` revision forks, the children of the fork will be merged. However, the `wobbler-stable` and `wobbler-unstable` branches will not be merged together, despite having a common ancestor.



For each branch, the set of revisions with *no children* is called the *heads* of the branch. Monotone can automatically locate, and attempt to merge, the heads of a branch. If it fails to automatically merge the heads, it may ask you for assistance or else fail cleanly, leaving the branch alone.

For example, if a fork's left child has a child of its own (a "left grandchild"), monotone will merge the fork's right child with the left grandchild, since those revisions are the heads of the branch. It will not merge the left child with the right child, because the left child is not a member of the heads.



When there is only one revision in the heads of a branch, we say that *the heads are merged*, or more generally that *the branch is merged*, since the heads is the logical set of candidates for any merging activity. If there are two or more revisions in the heads of a branch, and you ask to merge the branch, monotone will merge them two-at-a-time until there is only one.

1.7.1 Branch Names

The branch names used in the above section are fine for an example, but they would be bad to use in a real project. The reason is, monotone branch names must be *globally* unique, over all branches in the world. Otherwise, bad things can happen. Fortunately, we have a handy source of globally unique names — the DNS system.

When naming a branch, always prepend the reversed name of a host that you control or are otherwise authorized to use. For example, monotone development happens on the branch `net.venge.monotone`, because `venge.net` belongs to monotone's primary author. The idea is that this way, you can coordinate with other people using a host to make sure there are no conflicts — in the example, monotone's primary author can be certain that no-one else using `venge.net` will start up a different program named `monotone`. If you work for Yoyodyne, Inc. (owners of yoyodyne.com), then all your branch names should look like `com.yoyodyne.`*`something`*.

What the *`something`* part looks like is up to you, but usually the first part is the project name (the `monotone` in `net.venge.monotone`), and then possibly more stuff after that to describe a particular branch. For example, monotone's win32 support was initially developed on the branch `net.venge.monotone.win32`.

(For more information, see Section 3.9 [Naming Conventions], page 56.)

2 Tutorial

This chapter illustrates the basic uses of monotone by means of an example, fictional software project.

2.1 Issues

Before we walk through the tutorial, there are two minor issues to address: standard options and revision selectors.

2.1.1 Standard Options

Before operating monotone, two important command-line options should be explained.

- Most commands operate on a *database*, which is selected with the '`--db`' option.
- Many commands operate on a subset of the database, called a *branch*, which is selected with the '`--branch`' option.

Monotone will cache the settings for these options in your working copy, so ordinarily once you have checked out a project, you will not need to specify them again. We will therefore only mention these arguments in the first example.

2.1.2 Revision Selectors

Many commands require you to supply 40-character SHA1 values as arguments, which identify revisions. These "revision IDs" are tedious to type, so monotone permits you to supply "revision selectors" rather than complete revision IDs. Selectors are a more "human friendly" way of specifying revisions by combining certificate values into unique identifiers. This "selector" mechanism can be used anywhere a revision ID would normally be used. For details on selector syntax, see Section 3.1 [Selectors], page 44.

We are now ready to explore our fictional project.

2.2 The Fictional Project

Our fictional project involves 3 programmers cooperating to write firmware for a robot, the JuiceBot 7, which dispenses fruit juice. The programmers are named Jim, Abe and Beth.

- Jim lives in Japan, and owns JuiceBot Inc. You will know when we're talking about Jim, because everything he does involves the letter "j".
- Abe lives in Australia and writes code related to apple juice. You will know when we're talking about Abe, because everything he does involves the letter "a".
- Beth lives in Brazil and writes code related to banana juice. You will know when we're talking about Beth, because everything she does involves the letter "b".

In our example the programmers work privately on laptops, and are usually *disconnected* from the network. They share no storage system. Thus when each programmer enters a command, it affects only his or her own computer, unless otherwise stated.

In the following, our fictional project team will work through several version control tasks. Some tasks must be done by each member of our example team; other tasks involve only one member.

2.3 Creating a Database

The first step Jim, Abe and Beth each need to perform is to create a new database. This is done with the `monotone db init` command, providing a '`--db`' option to specify the location of the new database. Each programmer creates their own database, which will reside in their home directory and store all the revisions, files and manifests they work on. Monotone requires this step as an explicit command, to prevent spurious creation of databases when an invalid '`--db`' option is given.

In real life, most people prefer to keep one database for each project they work on. If we followed that convention here in the tutorial, though, then all the databases would be called `juicebot.db`, and that would make things more confusing to read. So instead, we'll have them each name their database after themselves.

Thus Jim issues the command:

```
$ monotone db init --db=~/jim.db
```

Abe issues the command:

```
$ monotone db init --db=~/abe.db
```

And Beth issues the command:

```
$ monotone db init --db=~/beth.db
```

2.4 Generating Keys

Now Jim, Abe and Beth must each generate an RSA key pair for themselves. This step requires choosing a key identifier. Typical key identifiers are similar to email addresses, possibly modified with some prefix or suffix to distinguish multiple keys held by the same owner. Our example programmers will use their email addresses at the fictional "juicebot.co.jp" domain name. When we ask for a key to be generated, monotone will ask us for a passphrase. This phrase is used to encrypt the key when storing it on disk, as a security measure.

Jim does the following:

```
$ monotone --db=~/jim.db genkey jim@juicebot.co.jp
monotone: generating key-pair 'jim@juicebot.co.jp'
enter passphrase for key ID [jim@juicebot.co.jp] : <Jim enters his passphrase>
confirm passphrase for key ID [jim@juicebot.co.jp]: <Jim confirms his passphrase>
monotone: storing key-pair 'jim@juicebot.co.jp' in database
```

Abe does something similar:

```
$ monotone --db=~/abe.db genkey abe@juicebot.co.jp
monotone: generating key-pair 'abe@juicebot.co.jp'
enter passphrase for key ID [abe@juicebot.co.jp] : <Abe enters his passphrase>
confirm passphrase for key ID [abe@juicebot.co.jp]: <Abe confirms his passphrase>
monotone: storing key-pair 'abe@juicebot.co.jp' in database
```

as does Beth:

```
$ monotone --db=~/beth.db genkey beth@juicebot.co.jp
monotone: generating key-pair 'beth@juicebot.co.jp'
enter passphrase for key ID [beth@juicebot.co.jp] : <Beth enters her passphrase>
confirm passphrase for key ID [beth@juicebot.co.jp]: <Beth confirms her passphrase>
monotone: storing key-pair 'beth@juicebot.co.jp' in database
```

Each programmer has now generated a key pair and placed it in their local database. Each can list the keys in their database, to ensure the correct key was generated. For example, Jim might see this:

```
$ monotone --db=~/jim.db list keys

[public keys]
9e9e9ef1d515ad58bfaa5cf282b4a872d8fda00c jim@juicebot.co.jp


[private keys]
771ace046c27770a99e5fddfa99c9247260b5401 jim@juicebot.co.jp
```

The hexadecimal string printed out before each key name is a *fingerprint* of the key, and can be used to verify that the key you have stored under a given name is the one you intended to store. Monotone will never permit one database to store two keys with the same name or the same fingerprint.

This output shows one private and one public key stored under the name `jim@juicebot.co.jp`, so it indicates that Jim's key-pair has been successfully generated and stored. On subsequent commands, Jim will need to re-enter his passphrase in order to perform security-sensitive tasks. Jim isn't very worried about security (and, more importantly, it simplifies the tutorial text to skip the passphrase prompts) so he decides to store his passphrase in his `monotonerc` file. He does this by writing a *hook function* which returns the passphrase:

```
$ mkdir ~/.monotone
$ cat >>~/.monotone/monotonerc
function get_passphrase(keypair_id)
  return "jimsekret"
end
^D
```

Now whenever monotone needs his passphrase, it will call this function instead of prompting him to type it. Note that we are appending the new hook to the (possibly existing) file. We do this to avoid losing other changes by mistake; therefore, be sure to check that no other `get_passphrase` function appears in the configuration file.

Abe and Beth do the same, with their secret passphrases.

2.5 Starting a New Project

Before he can begin work on the project, Jim needs to create a *working copy* — a directory whose contents monotone will keep track of. Often, one works on projects that someone else has started, and creates working copies with the `checkout` command, which you'll learn about later. Jim is starting a new project, though, so he does something a little bit different. He uses the `monotone setup` command to create a new working copy.

This command creates the named directory (if it doesn't already exist), and creates the '`MT`' directory within it. The '`MT`' directory is how monotone recognizes that a directory is a working copy, and monotone stores some bookkeeping files within it. For instance, command line values for the '`--db`', '`--branch`' or '`--key`' options to the `setup` command will be cached in a file called '`MT/options`', so you don't have to keep passing them to monotone all the time.

He chooses `jp.co.juicebot.jb7` as a branch name. (See Section 3.9 [Naming Conventions], page 56 for more information about appropriate branch names.) Jim then creates his working copy:

```
/home/jim$ monotone --db=jim.db --branch=jp.co.juicebot.jb7 setup juice
/home/jim$ cd juice
/home/jim/juice$
```

Notice that Jim has changed his current directory to his newly created working copy. For the rest of this example we will assume that everyone issues all further monotone commands from their working copy directories.

2.6 Adding Files

Next Jim decides to add some files to the project. He writes up a file containing the prototypes for the JuiceBot 7:

```
$ mkdir include
$ cat >include/jb.h
/* Standard JuiceBot hw interface */

#define FLOW_JUICE 0x1
#define POLL_JUICE 0x2
int spoutctl(int port, int cmd, void *x);

/* JuiceBot 7 API */

#define APPLE_SPOUT 0x7e
#define BANANA_SPOUT 0x7f
void dispense_apple_juice ();
void dispense_banana_juice ();
^D
```

Then adds a couple skeleton source files which he wants Abe and Beth to fill in:

```
$ mkdir src
$ cat >src/apple.c
#include "jb.h"

void
dispense_apple_juice()
{
  /* Fill this in please, Abe. */
}
^D
$ cat >src/banana.c
#include "jb.h"

void
dispense_banana_juice()
{
  /* Fill this in please, Beth. */
}
^D
```

Now Jim tells monotone to add these files to its record of his working copy. He specifies one filename and one directory; monotone recursively scans the directory and adds all its files.

```
$ monotone add include/jb.h src
monotone: adding include/jb.h to working copy add set
monotone: adding src/apple.c to working copy add set
monotone: adding src/banana.c to working copy add set
```

This command produces a record of Jim's intentions in a special file called '`MT/work`', stored in the working copy. The file is plain text:

```
$ cat MT/work
add_file "include/jb.h"

add_file "src/apple.c"

add_file "src/banana.c"
```

Jim then gets up from his machine to get a coffee. When he returns he has forgotten what he was doing. He asks monotone:

```
$ monotone status

new_manifest [2098eddbe833046174de28172a813150a6cbda7b]

old_revision []
old_manifest []

add_file "include/jb.h"

add_file "src/apple.c"

add_file "src/banana.c"

patch "include/jb.h"
 from []
   to [3b12b2d0b31439bd50976633db1895cff8b19da0]

patch "src/apple.c"
 from []
   to [2650ffc660dd00a08b659b883b65a060cac7e560]

patch "src/banana.c"
 from []
   to [e8f147e5b4d5667f3228b7bba1c5c1e639f5db9f]
```

The output of this command tells Jim that his edits, so far, constitute only the addition of some files. In the output we can see one pecularity of monotone's changeset format. The pecularity is that when monotone records a "new file", it actually records two separate events: the addition of an empty file to the working copy, and a patch of that file from empty to its intended contents.

Jim wants to see the actual details of the files he added, however, so he runs a command which prints out the status *and* a GNU "unified diff" of the patches involved in the changeset:

```
$ monotone diff
#
# add_file "include/jb.h"
#
# add_file "src/apple.c"
#
# add_file "src/banana.c"
#
# patch "include/jb.h"
#  from []
#    to [3b12b2d0b31439bd50976633db1895cff8b19da0]
#
# patch "src/apple.c"
#  from []
#    to [2650ffc660dd00a08b659b883b65a060cac7e560]
#
# patch "src/banana.c"
#  from []
#    to [e8f147e5b4d5667f3228b7bba1c5c1e639f5db9f]
#
============================================================================
--- include/jb.h
+++ include/jb.h 3b12b2d0b31439bd50976633db1895cff8b19da0
@ -0,0 +1,13 @
+/* Standard JuiceBot hw interface */
+
+#define FLOW_JUICE 0x1
+#define POLL_JUICE 0x2
+#define SET_INTR 0x3
+int spoutctl(int port, int cmd, void *x);
+
+/* JuiceBot 7 API */
+
+#define APPLE_SPOUT 0x7e
+#define BANANA_SPOUT 0x7f
+void dispense_apple_juice ();
+void dispense_banana_juice ();
============================================================================
--- src/apple.c
+++ src/apple.c 2650ffc660dd00a08b659b883b65a060cac7e560
@ -0,0 +1,7 @
+#include "jb.h"
+
+void
+dispense_apple_juice()
+{
+  /* Fill this in please, Abe. */
+}
============================================================================
--- src/banana.c
+++ src/banana.c e8f147e5b4d5667f3228b7bba1c5c1e639f5db9f
@ -0,0 +1,7 @
+#include "jb.h"
+
+void
+dispense_banana_juice()
+{
+  /* Fill this in please, Beth. */
+}
```

2.7 Committing Work

Satisfied with the work he's done, Jim wants to save his changes. He then commits his working copy, which causes monotone to process the '`MT/work`' file and record the file contents, manifest, and revision into the database. Since he provided a branch name when he ran `setup`, monotone will use this as the default branch name when he commits.

```
$ monotone commit --message="initial checkin of project"
monotone: beginning commit on branch 'jp.co.juicebot.jb7'
monotone: committed revision 2e24d49a48adf9acf3a1b6391a080008cbef9c21
```

When monotone committed Jim's revision, erased the '`MT/work`' file, and wrote a new file called '`MT/revision`', which contains the working copy's new base revision ID. Jim can use this revision ID in the future, as an argument to the `checkout` command, if he wishes to return to this revision:

```
$ cat MT/revision
2e24d49a48adf9acf3a1b6391a080008cbef9c21
```

Monotone also generated a number of certificates, attached to the new revision. These certs store metadata about the commit. Jim can ask monotone for a list of certs on this revision.

```
$ monotone ls certs 2e24d49a48adf9acf3a1b6391a080008cbef9c21
-----------------------------------------------------------------
Key   : jim@juicebot.co.jp
Sig   : ok
Name  : branch
Value : jp.co.juicebot.jb7
-----------------------------------------------------------------
Key   : jim@juicebot.co.jp
Sig   : ok
Name  : date
Value : 2004-10-26T02:53:08
-----------------------------------------------------------------
Key   : jim@juicebot.co.jp
Sig   : ok
Name  : author
Value : jim@juicebot.co.jp
-----------------------------------------------------------------
Key   : jim@juicebot.co.jp
Sig   : ok
Name  : changelog
Value : initial checkin of project
```

The output of this command has a block for each cert found. Each block has 4 significant pieces of information. The first indicates the signer of the cert, in this case `jim@juicebot.co.jp`. The second indicates whether this cert is "ok", meaning whether the RSA signature provided is correct for the cert data. The third is the cert name, and the fourth is the cert value. This list shows us that monotone has confirmed that, according to `jim@juicebot.co.jp`, the revision `2e24d49a48adf9acf3a1b6391a080008cbef9c21` is a member of the branch `jp.co.juicebot.jb7`, written by `jim@juicebot.co.jp`, with the given date and changelog.

It is important to keep in mind that revisions are not "in" or "out" of a branch in any global sense, nor are any of these cert values *true* or *false* in any global sense. Each cert indicates that *some person* – in this case Jim – would like to associate a revision with some value; it is up to you to decide if you want to accept that association.

Jim can now check the status of his branch using the “heads” command, which lists all the head revisions in the branch:

```
$ monotone heads
branch 'jp.co.juicebot.jb7' is currently merged:
2e24d49a48adf9acf3a1b6391a080008cbef9c21 jim@juicebot.co.jp 2004-10-26T02:53:08
```

The output of this command tells us that there is only one current “head” revision in the branch `jp.co.juicebot.jb7`, and it is the revision Jim just committed. A head revision is one without any descendents. Since Jim has not committed any changes to this revision yet, it has no descendents.

2.8 Network Service

Jim now decides he will make his base revision available to his employees. To do this he gives Abe and Beth permission to access his database. There are two parts to this: first, he has to get a copy of each of their public keys; then, he has to tell monotone that the holders of those keys are permitted to access his database.

First, Abe exports his public key:

```
$ monotone --db=~/abe.db pubkey abe@juicebot.co.jp >~/abe.pubkey
```

His public key is just a plain block of ASCII text:

```
$ cat ~/abe.pubkey
[pubkey abe@juicebot.co.jp]
MIGdMA0GCSqGSIb3DQEBAQUAA4GLADCBhwKBgQCbaVff9SF78FiB/1nUdmjbU/TtPyQqe/fW
CDg7hSg1yY/hWgClXE9FI0bHtjPMIx1kBOig09AkCT7tBXM9z6iGWxTBhSR7D/qsJQGPorOD
DO7xovIHthMbZZ9FnvyB/BCyiibdWgGT0Gtq94OKdvCRNuT59e5v9L4pBkvajb+IzQIBEQ==
[end]
```

Beth also exports her public key:

```
$ monotone --db=~/beth.db pubkey beth@juicebot.co.jp >~/beth.pubkey
```

Then Abe and Beth both send their keys to Jim. The keys are not secret, but the team members must be relatively certain that they are communicating with the person they intend to trust, when exchanging keys, and not some malicious person pretending to be a team member. Key exchange may involve sending keys over an encrypted medium, or meeting in person to exchange physical copies, or any number of techniques. All that matters, ultimately, is that Jim receives both Abe's and Beth's key.

So eventually, after key exchange, Jim has the public key files in his home directory. He tells monotone to read the associated key packets into his database:

```
$ cat ~/abe.pubkey ~/beth.pubkey | monotone --db=~/jim.db read
monotone: read 2 packets
```

Now Jim's monotone is able to identify Beth and Abe, and he is ready to give them permission to access his database. He does this by adding a small amount of extra information to his '`monotonerc`' file:

```
$ cat >>~/.monotone/monotonerc
function get_netsync_read_permitted (branch, identity)
  if (identity == "abe@juicebot.co.jp") then return true end
  if (identity == "beth@juicebot.co.jp") then return true end
  return false
end

function get_netsync_write_permitted (identity)
  if (identity == "abe@juicebot.co.jp") then return true end
  if (identity == "beth@juicebot.co.jp") then return true end
  return false
end
^D
```

He then makes sure that his TCP port 5253 is open to incoming connections, adjusting his firewall settings as necessary, and runs the monotone `serve` command:

```
$ monotone --db=jim.db serve jim-laptop.juicebot.co.jp "jp.co.juicebot.jb7*"
```

This command sets up a single listener loop on the host `jim-laptop.juicebot.co.jp`, serving all branches matching `jp.co.juicebot.jb7*`. This will naturally include the `jp.co.juicebot.jb7` branch, and any sub-branches. The quotes around `"jp.co.juicebot.jb7*"` are there to protect the `*` from expansion by the shell; they have no meaning to monotone.

Now Abe decides he wishes to fetch Jim's code. To do this he issues the monotone `sync` command:

```
$ monotone --db=abe.db sync jim-laptop.juicebot.co.jp "jp.co.juicebot.jb7*"
monotone: setting default server to jim-laptop.juicebot.co.jp
monotone: setting default branch include pattern to 'jp.co.juicebot.jb7*'
monotone: setting default branch exclude pattern to ''
monotone: connecting to jim-laptop.juicebot.co.jp
monotone: first time connecting to server jim-laptop.juicebot.co.jp:5253
monotone: I'll assume it's really them, but you might want to double-check
monotone: their key's fingerprint: 9e9e9ef1d515ad58bfaa5cf282b4a872d8fda00c
monotone: warning: saving public key for jim@juicebot.co.jp to database
monotone: finding items to synchronize:
monotone: bytes in | bytes out | revs in | revs out | revs written
monotone:     2587 |      1025 |       1 |        0 |            1
monotone: successful exchange with jim-laptop.juicebot.co.jp
```

Abe now has, in his database, a copy of everything Jim put in the branch. Therefore Abe can disconnect from the expensive network connection he's on and work locally for a while. When Abe wants to send work back to Jim, or get new work Jim has added, all he needs to do is run the `sync` command again and work will flow both ways, bringing each party up to date with the work of the other.

At this point Jim is operating as a sort of "central server" for the company. If Jim wants to, he can leave his server running forever, or even put his server on a dedicated computer with better network connectivity. But if Jim is ever unable to play this role of "central server", perhaps due to a network failure, either Beth or Abe can run the `serve` command and provide access for the other to `sync` with. In fact, each employee can run a server if they like, concurrently, to help minimize the risk of service disruption from hardware failures. Changes will flow between servers automatically as clients access them and trade with one another.

In practice, most people like to use at least one central server that is always running; this way, everyone always knows where to go to get the latest changes, and people can push their changes out without first calling their friends and making sure that they have their servers running. To make this style of working more convenient, monotone remembers the first server you use, and makes that the default for future operations.

2.9 Making Changes

Abe decides to do some work on his part of the code. He has a copy of Jim's database contents, but cannot edit any of that data yet. He begins his editing by checking out the head of the `jp.co.juicebot.jb7` branch into a working copy, so he can edit it:

```
$ monotone --db=abe.db --branch=jp.co.juicebot.jb7 checkout .
```

Monotone unpacks the set of files in the head revision's manifest directly into Abe's current directory. (If he had specified something other than '`.`' at the end, monotone would have created that directory and unpacked the files into it.) Abe then opens up one of the files, '`src/apple.c`', and edits it:

```
$ vi src/apple.c
<Abe writes some apple-juice dispensing code>
```

The file '`src/apple.c`' has now been *changed*. Abe gets up to answer a phone call, and when he returns to his work he has forgotten what he changed. He can ask monotone for details:

```
$ monotone diff
#
# patch "src/apple.c"
#  from [2650ffc660dd00a08b659b883b65a060cac7e560]
#    to [e2c418703c863eabe70f9bde988765406f885fd0]
#
============================================================================
--- src/apple.c 2650ffc660dd00a08b659b883b65a060cac7e560
+++ src/apple.c e2c418703c863eabe70f9bde988765406f885fd0
@ -1,7 +1,10 @
 #include "jb.h"

 void
 dispense_apple_juice()
 {
-  /* Fill this in please, Abe. */
+  spoutctl(APPLE_SPOUT, FLOW_JUICE, 1);
+  while (spoutctl(APPLE_SPOUT, POLL_JUICE, 1) == 0)
+    usleep (1000);
+  spoutctl(APPLE_SPOUT, FLOW_JUICE, 0);
 }
```

Satisfied with his day's work, Abe decides to commit.

```
$ monotone commit
monotone: beginning commit on branch 'jp.co.juicebot.jb7'
```

Abe neglected to provide a '`--message`' option specifying the change log on the command line and the file '`MT/log`' is empty because he did not document his changes there. Monotone therefore invokes an external "log message editor" — typically an editor like `vi` — with an explanation of the changes being committed and the opportunity to enter a log message.

```
polling implementation of src/apple.c
MT:
MT: ----------------------------------------------------------------------
MT: Enter Log.  Lines beginning with ‘MT:’ are removed automatically
MT:
MT: new_manifest [b33cb337dccf21d6673f462d677a6010b60699d1]
MT:
MT: old_revision [2e24d49a48adf9acf3a1b6391a080008cbef9c21]
MT: old_manifest [2098eddbe833046174de28172a813150a6cbda7b]
MT:
MT: patch "src/apple.c"
MT: from [2650ffc660dd00a08b659b883b65a060cac7e560]
MT:   to [e2c418703c863eabe70f9bde988765406f885fd0]
MT:
MT: ----------------------------------------------------------------------
MT:
```

Abe enters a single line above the explanatory message, saying “polling implementation of src/apple.c”. He then saves the file and quits the editor. Monotone deletes all the lines beginning with “MT:” and leaves only Abe’s short message. Returning to the shell, Abe’s commit completes:

```
monotone: committed revision 70decb4b31a8227a629c0e364495286c5c75f979
```

Abe then sends his new revision back to Jim:

```
$ monotone sync
monotone: connecting to jim-laptop.juicebot.co.jp
monotone: finding items to synchronize:
monotone:   certs |    keys | revisions
monotone:       8 |       2 |         2
monotone: bytes in | bytes out | revs in | revs out | revs written
monotone:      615 |      2822 |       0 |        1 |            0
monotone: successful exchange with jim-laptop.juicebot.co.jp
```

Beth does a similar sequence. First she syncs her database with Jim’s:

```
$ monotone --db=beth.db sync jim-laptop.juicebot.co.jp "jp.co.juicebot.jb7*"
monotone: setting default server to jim-laptop.juicebot.co.jp
monotone: setting default branch include pattern to ’jp.co.juicebot.jb7*’
monotone: setting default branch exclude pattern to ’’
monotone: connecting to jim-laptop.juicebot.co.jp
monotone: first time connecting to server jim-laptop.juicebot.co.jp:5253
monotone: I’ll assume it’s really them, but you might want to double-check
monotone: their key’s fingerprint: 9e9e9ef1d515ad58bfaa5cf282b4a872d8fda00c
monotone: warning: saving public key for jim@juicebot.co.jp to database
monotone: finding items to synchronize:
monotone: bytes in | bytes out | revs in | revs out | revs written
monotone:     4601 |      1239 |       2 |        0 |            1
monotone: verifying new revisions (this may take a while)
monotone: bytes in | bytes out | revs in | revs out | revs written
monotone:     4601 |      1285 |       2 |        0 |            2
monotone: successful exchange with jim-laptop.juicebot.co.jp
```

She checks out a copy of the tree from her database:

```
$ monotone --db=beth.db --branch=jp.co.juicebot.jb7 checkout .
```

She edits the file ‘`src/banana.c`’:

```
$ vi src/banana.c
<Beth writes some banana-juice dispensing code>
```

and logs her changes in ‘`MT/log`’ right away so she does not forget what she has done like Abe.

```
$ vi MT/log
* src/banana.c: Added polling implementation
```

Later, she commits her work. Monotone again invokes an external editor for her to edit her log message, but this time it fills in the messages she's written so far, and she simply checks them over one last time before finishing her commit:

```
$ monotone commit
monotone: beginning commit on branch 'jp.co.juicebot.jb7'
monotone: committed revision 80ef9c9d251d39074d37e72abf4897e0bbae1cfb
```

And she syncs with Jim again:

```
$ monotone sync
monotone: connecting to jim-laptop.juicebot.co.jp
monotone: finding items to synchronize:
monotone:   certs |    keys | revisions
monotone:      12 |       3 |         3
monotone: bytes in | bytes out | revs in | revs out | revs written
monotone:      709 |      2879 |       0 |        1 |            0
monotone: successful exchange with jim-laptop.juicebot.co.jp
```

2.10 Dealing with a Fork

Careful readers will note that, in the previous section, the JuiceBot company's work was perfectly serialized:

1. Jim did some work
2. Abe synced with Jim
3. Abe did some work
4. Abe synced with Jim
5. Beth synced with Jim
6. Beth did some work
7. Beth synced with Jim

The result of this ordering is that Jim's work entirely preceeded Abe's work, which entirely preceeded Beth's work. Moreover, each worker was fully informed of the "up-stream" worker's actions, and produced purely derivative, "down-stream" work:

1. Jim made revision 2e24d...
2. Abe changed revision 2e24d... into revision 70dec...
3. Beth derived revision 70dec... into revision 80ef9...

This is a simple, but sadly unrealistic, ordering of events. In real companies or work groups, people often work in parallel, *diverging* from commonly known revisions and *merging* their work together, sometime after each unit of work is complete.

Monotone supports this diverge/merge style of operation naturally; any time two revisions diverge from a common parent revision, we say that the revision graph has a *fork* in it. Forks can happen at any time, and require no coordination between workers. In fact any interleaving of the previous events would work equally well; with one exception: if forks were produced, someone would eventually have to run the `merge` command, and possibly resolve any conflicts in the fork.

To illustrate this, we return to our workers Beth and Abe. Suppose Jim sends out an email saying that the current polling juice dispensers use too much CPU time, and must be rewritten to use the JuiceBot's interrupt system. Beth wakes up first and begins working immediately, basing her work off the revision 80ef9... which is currently in her working copy:

```
$ vi src/banana.c
<Beth changes her banana-juice dispenser to use interrupts>
```

Beth finishes and examines her changes:

```
$ monotone diff
#
# patch "src/banana.c"
#  from [7381d6b3adfddaf16dc0fdb05e0f2d1873e3132a]
#    to [5e6622cf5c8805bcbd50921ce7db86dad40f2ec6]
#
============================================================================
--- src/banana.c 7381d6b3adfddaf16dc0fdb05e0f2d1873e3132a
+++ src/banana.c 5e6622cf5c8805bcbd50921ce7db86dad40f2ec6
@ -1,10 +1,15 @
 #include "jb.h"

+static void
+shut_off_banana()
+{
+  spoutctl(BANANA_SPOUT, SET_INTR, 0);
+  spoutctl(BANANA_SPOUT, FLOW_JUICE, 0);
+}
+
 void
-dispense_banana_juice()
+dispense_banana_juice()
 {
+  spoutctl(BANANA_SPOUT, SET_INTR, &shut_off_banana);
   spoutctl(BANANA_SPOUT, FLOW_JUICE, 1);
-  while (spoutctl(BANANA_SPOUT, POLL_JUICE, 1) == 0)
-    usleep (1000);
-  spoutctl(BANANA_SPOUT, FLOW_JUICE, 0);
 }
```

She commits her work:

```
$ monotone commit --message="interrupt implementation of src/banana.c"
monotone: beginning commit on branch 'jp.co.juicebot.jb7'
monotone: committed revision 8b41b5399a564494993063287a737d26ede3dee4
```

And she syncs with Jim:

```
$ monotone sync
```

Unfortunately, before Beth managed to sync with Jim, Abe had woken up and implemented a similar interrupt-based apple juice dispenser, but his working copy is 70dec..., which is still “upstream” of Beth’s.

```
$ vi apple.c
<Abe changes his apple-juice dispenser to use interrupts>
```

Thus when Abe commits, he unknowingly creates a fork:

```
$ monotone commit --message="interrupt implementation of src/apple.c"
```

Abe does not see the fork yet; Abe has not actually seen *any* of Beth’s work yet, because he has not synchronized with Jim. Since he has new work to contribute, however, he now syncs:

```
$ monotone sync
```

Now Jim and Abe will be aware of the fork. Jim sees it when he sits down at his desk and asks monotone for the current set of heads of the branch:

```
$ monotone heads
monotone: branch 'jp.co.juicebot.jb7' is currently unmerged:
39969614e5a14316c7ffefc588771f491c709152 abe@juicebot.co.jp 2004-10-26T02:53:16
8b41b5399a564494993063287a737d26ede3dee4 beth@juicebot.co.jp 2004-10-26T02:53:15
```

Clearly there are two heads to the branch: it contains an un-merged fork. Beth will not yet know about the fork, but in this case it doesn't matter: anyone can merge the fork, and since there are no conflicts Jim does so himself:

```
$ monotone merge
monotone: starting with revision 1 / 2
monotone: merging with revision 2 / 2
monotone: [source] 39969614e5a14316c7ffefc588771f491c709152
monotone: [source] 8b41b5399a564494993063287a737d26ede3dee4
monotone: common ancestor 70decb4b31a8227a629c0e364495286c5c75f979 abe@juicebot.co.jp  2004-
10-26T:02:50:01 found
monotone: trying 3-way merge
monotone: [merged] da499b9d9465a0e003a4c6b2909102ef98bf4e6d
monotone: your working copies have not been updated
```

The output of this command shows Jim that two heads were found, combined via a 3-way merge with their ancestor, and saved to a new revision. This happened automatically, because the changes between the common ancestor and heads did not conflict. If there had been a conflict, monotone would have invoked an external merging tool to help resolve it.

After merging, the branch has a single head again, and Jim updates his working copy.

```
$ monotone update
monotone: selected update target da499b9d9465a0e003a4c6b2909102ef98bf4e6d
monotone: updating src/apple.c to f088e24beb43ab1468d7243e36ce214a559bdc96
monotone: updating src/banana.c to 5e6622cf5c8805bcbd50921ce7db86dad40f2ec6
monotone: updated to base revision da499b9d9465a0e003a4c6b2909102ef98bf4e6d
```

The update command selected an update target — in this case the newly merged head — and performed an in-memory merge between Jim's working copy and the chosen target. The result was then written to Jim's working copy. If Jim's working copy had any uncommitted changes in it, they would have been merged with the update in exactly the same manner as the merge of multiple committed heads.

Monotone makes very little distinction between a "pre-commit" merge (an update) and a "post-commit" merge. Both sorts of merge use the exact same algorithm. The major difference concerns the recoverability of the pre-merge state: if you commit your work first, and merge after committing, then even if the merge somehow fails (due to difficulty in a manual merge step, for instance), your committed state is still safe. If you update, on the other hand, you are requesting that monotone directly modify your working copy, and while monotone will try hard not to break anything, this process is inherently more open to error. It is therefore recommended that you commit your work *first*, before merging.

If you have previously used another version control system, this may at first seem surprising; there are some systems where you are *required* to update, and risk the above problems, before you can commit. Monotone, however, was designed with this problem in mind, and thus *always* allows you to commit before merging. A good rule of thumb is to only use `update` in working copies with no local modifications, or when you actually want to work against a different base revision (perhaps because finishing your change turns out to require some fixes made in another revision, or because you discover that you have accidentally started working against a revision that contains unrelated bugs, and need to back out to a working revision for testing).

2.11 Branching and Merging

So by now you're familiar with making changes, sharing them with other people, and integrating your changes with their changes. Sometimes, though, you may want to make some changes, and *not* integrate them with other people's — or at least not right away. One way to do this would be to simply never run `monotone merge`; but it would quickly become confusing to try and keep track of which changes were in which revisions. This is where *branches* are useful.

Continuing our example, suppose that Jim is so impressed by Beth's work on banana juice support that he assigns her to work on the JuiceBot 7's surprise new feature: muffins. In the mean time, Abe will continue working on the JuiceBot's basic juice-related functions.

The changes required to support muffins are somewhat complicated, and Beth is worried that her work might destabilize the program, and interfere with Abe's work. In fact, she isn't even sure her first attempt will turn out to be the right approach; she might work on it for a while and then decide it was a bad idea, and should be discarded. For all these reasons, she decides that she will work on a branch, and then once she is satisfied with the new code, she will merge back onto the mainline.

She decides that since main development is in branch `jp.co.juicebot.jb7`, she will use branch `jp.co.juicebot.jb7.muffins`. So, she makes the first few edits to the new muffins code, and commits it on a new branch by simply passing '`--branch`' to commit:

```
$ monotone commit --branch=jp.co.juicebot.jb7.muffins --message='autobake framework'
monotone: beginning commit on branch 'jp.co.juicebot.jb7.muffins'
monotone: committed revision d33caefd61823ecbb605c39ffb84705dec449857
```

That's all there is to it — there is now a `jp.co.juicebot.jb7.muffins` branch, with her initial checkin on it. She can make further checkins from the same working copy, and they will automatically go to the muffins branch; if anyone else wants to help her work on muffins, they can check out that branch as usual.

Of course, while Beth is working on the new muffins code, Abe is still making fixes to the main line. Occasionally, Beth wants to integrate his latest work into the muffins branch, so that her version doesn't fall too far behind. She does this by using the `propagate` command:

```
$ monotone propagate jp.co.juicebot.jb7 jp.co.juicebot.jb7.muffins
monotone: propagating jp.co.juicebot.jb7 -> jp.co.juicebot.jb7.muffins
monotone: [source] da003f115752ac6e4750b89aaca9dbba178ac80c
monotone: [target] d0e5c93bb61e5fd25a0dadf41426f209b73f40af
monotone: common ancestor 853b8c7ac5689181d4b958504adfb5d07fd959ab jim@juicebot.co.jp 2004-
10-26T:12:44:23 found
monotone: trying 3-way merge
monotone: [merged] 89585b3c5e51a5a75f5d1a05dda859c5b7dde52f
```

The `propagate` merges all of the new changes on one branch onto another.

When the muffins code is eventually stable and ready to be integrated into the main line of development, she simply propagates the other way:

```
$ monotone propagate jp.co.juicebot.jb7.muffins jp.co.juicebot.jb7
monotone: propagating jp.co.juicebot.jb7.muffins -> jp.co.juicebot.jb7
monotone: [source] 4e48e2c9a3d2ca8a708cb0cc545700544efb5021
monotone: [target] bd29b2bfd07644ab370f50e0d68f26dcfd3bb4af
monotone: common ancestor 652b1035343281a0d2a5de79919f9a31a30c9028 jim@juicebot.co.jp 2004-
10-26T:15:25:05 found
monotone: [merged] 03f7495b51cc70b76872ed019d19dee1b73e89b6
```

Monotone always records the full history of all merges, and is designed to handle an arbitrarily complicated graph of changes. You can make a branch, then branch off from that branch, propagate changes between arbitrary branches, and so on; monotone will track all of it, and do something sensible for each merge. Of course, it is still probably a good idea to come up with some organization of branches and a plan for which should be merged to which other ones. Monotone may keep track of graphs of arbitrary complexity; but, you will have more trouble. Whatever arrangement of branches you come up with, though, monotone should be able to handle it.

3 Advanced Uses

This chapter covers slightly less common aspects of using monotone. Some users of monotone will find these helpful, though possibly not all. We assume that you have read through the taxonomy and tutorial, and possibly spent some time playing with the program to familiarize yourself with its operation.

3.1 Selectors

Revisions can be specified on the monotone command line, precisely, by entering the entire 40-character hexadecimal SHA1 code. This can be cumbersome, so monotone also allows a more general syntax called “selectors” which is less precise but more “human friendly”. Any command which expects a precise revision ID can also accept a selector in its place; in fact a revision ID is just a special type of selector which is very precise.

Simple examples

Some selector examples are helpful in clarifying the idea:

`a432`
: Revision IDs beginning with the string `a432`

`graydon@pobox.com/2004-04`
: Revisions written by `graydon@pobox.com` in April 2004.

`"jrh@example.org/2 weeks ago"`
: Revisions written by `jrh@example.org` 2 weeks ago.

`graydon/net.venge.monotone.win32/yesterday`
: Revisions in the `net.venge.monotone.win32` branch, written by `graydon`, yesterday.

A moment’s examination reveals that these specifications are “fuzzy” and indeed may return multiple values, or may be ambiguous. When ambiguity arises, monotone will inform you that more detail is required, and list various possibilities. The precise specification of selectors follows.

Selectors in detail

A selector is a combination of a selector type, which is a single xoASCII character, followed by a `:` character and a selector string. All selectors strings except for selector type `c` are just values. The value is matched against identifiers or certs, depending on its type, in an attempt to match a single revision. Selectors are matched as prefixes. The current set of selection types are:

Generic cert selector
: Uses selector type `c`. The selector string has the syntax *name* or *name*`=`*value*. The former syntax will select any revision that has a cert with that name, regardless of value; the latter will match any revision that has a cert with that name and value. Values to match for can have shell wildcards.

Author selection
: Uses selector type `a`. For example, `a:graydon` matches `author` certs where the cert value contains `graydon`.

Branch selection
: Uses selector type `b`. For example, `b:net.venge.monotone` matches `branch` certs where the cert value is `net.venge.monotone`. Values to match for can have shell wildcards.

Date selection
: Uses selector type `d`. For example, `d:2004-04` matches `date` certs where the cert value begins with `2004-04`. This selector also accepts expanded date syntax (see below).

"Earlier or equal than" selection
: Uses selector type `e`. For example, `e:2004-04-25` matches `date` certs where the cert value is less or equal than `2004-04-25T00:00:00`. If the time component is unspecified, monotone will assume 00:00:00. This selector also accepts expanded date syntax (see below)

"Later than" selection
: Uses selector type `l`. For example, `l:2004-04-25` matches `date` certs where the cert value is strictly less than `2004-04-25T00:00:00`. If the time component is unspecified, monotone will assume 00:00:00. This selector also accepts expanded date syntax (see below)

Identifier selection
: Uses selector type `i`. For example, `i:0f3a` matches revision IDs which begin with `0f3a`.

Tag selection
: Uses selector type `t`. For example, `t:monotone-0.11` matches `tag` certs where the cert value begins with `monotone-0.11`. Values to match for can have shell wildcards.

Further selector types may be added in the future.

Composite selectors

Selectors may be combined with the `/` character. The combination acts as database intersection (or logical `and`). For example, the selector `a:graydon/d:2004-04` can be used to select a revision which has an `author` cert beginning with `graydon` *as well as* a `date` cert beginning with `2004-04`.

Selector expansion

Before selectors are passed to the database, they are expanded using a lua hook: `expand_selector`. The default definition of this hook attempts to guess a number of common forms for selection, allowing you to omit selector types in many cases. For example, the hook guesses that the typeless selector `jrh@example.org` is an author selector, due to its syntactic form, so modifies it to read `a:jrh@example.org`. This hook will generally assign a selector type to values which “look like” partial hex strings, email addresses, branch names, or date specifications. For the complete source code of the hook, see Chapter 6 [Hook Reference], page 105.

Expanding dates

All date-related selectors (`d`, `e`, `l`) support an english-like syntax similar to CVS. This syntax is expanded to the numeric format by a lua hook: `expand_date`. The allowed date formats are:

now Expands to the current date and time.

today Expands to today's date. `e` and `l` selectors assume time 00:00:00

yesterday Expands to yesterday's date. `e` and `l` selectors assume time 00:00:00

<number> {minute|hour} <ago>
: Expands to today date and time, minus the specified `number` of minutes|hours.

<number> {day|week|month|year} <ago>
: Expands to today date, minus the specified `number` of days|weeks|months|years. `e` and `l` selectors assume time 00:00:00

<year>-<month>[-day[Thour:minute:second]]
: Expands to the supplied year/month. The day and time component are optional. If missing, `e` and `l` selectors assume the first day of month and time 00:00:00. The time component, if supplied, must be complete to the second.

For the complete source code of the hook, see Chapter 6 [Hook Reference], page 105.

Typeless selection

If, after expansion, a selector still has no type, it is matched as a special "unknown" selector type, which will match either a tag, an author, or a branch. This costs slightly more database access, but often permits simple selection using an author's login name and a date. For example, the selector `graydon/net.venge.monotone.win32/yesterday` would pass through the selector `graydon` as an unknown selector; so long as there are no branches or tags beginning with the string `graydon` this is just as effective as specifying `a:graydon`.

3.2 Restrictions

Several monotone commands accept optional *pathname...* arguments in order to establish a "restriction". Restrictions are used to limit the files and directories these commands examine for changes when comparing the working copy to the revision it is based on. Restricting a command to a specified set of files or directories simply ignores changes to files or directories not included by the restriction.

The following commands all support restrictions using optional *pathname...* arguments:

- `status`
- `diff`
- `revert`
- `commit`
- `list known`
- `list unknown`
- `list ignored`
- `list missing`

Including either the old or new name of a renamed file or directory will cause both names to be included in a restriction. If in doubt, the `status` command can be used to "test" a set of pathnames to ensure that the expected files are included or excluded by a restriction.

Commands which support restrictions also support the '`--depth=n`' option, where *n* specifies the maximum number of directories to descend. For example, *n*=0 disables recursion, *n*=1 means descend at most one directory, and so on.

The `update` command does not allow for updates to a restricted set of files, which may be slightly different than other version control systems. Partial updates don't really make sense in monotone, as they would leave the working copy based on a revision that doesn't exist in the database, starting an entirely new line of development.

Subdirectory restrictions

The restrictions facility also allows commands to operate from within a subdirectory of the working copy. By default, the *entire working copy* is always examined for changes. However, specifying an explicit `"."` pathname to a command will restrict it to the current subdirectory. Note that this is quite different from other version control systems and may seem somewhat surprising.

The expectation is that requiring a single `"."` to restrict to the current subdirectory should be simple to use. While the alternative, defaulting to restricting to the current subdirectory, would require a somewhat complicated ../../.. sequence to remove the restriction and operate on the whole tree.

This default was chosen because monotone versions whole project trees and generally expects to commit all changes in the working copy as a single atomic unit. Other version control systems often version individual files or directories and may not support atomic commits at all.

When working from within a subdirectory of the working copy all paths specified to monotone commands must be relative to the current subdirectory.

Finding a working copy

Monotone only stores a single 'MT' directory at the root of a working copy. Because of this, a search is done to find the 'MT' directory in case a command is executed from within a subdirectory of a working copy. Before a command is executed, the search for a working copy directory is done by traversing parent directories until an 'MT' directory is found or the filesystem root is reached. Upon finding an 'MT' directory, the 'MT/options' file is read for default options. The '--root' option may be used to stop the search early, before reaching the root of the physical filesystem.

Many monotone commands don't require a working copy and will simply proceed with no default options if no 'MT' directory is found. However, some monotone commands do require a working copy and will fail if no 'MT' directory can be found.

The `checkout` and `setup` commands create a *new working copy* and initialize a new 'MT/options' file based on their current option settings.

3.3 Scripting

People often want to write programs that call monotone — for example, to create a graphical interface to monotone's functionality, or to automate some task. For most programs, if you want to do this sort of thing, you just call the command line interface, and do some sort of parsing of the output. Monotone's output, however, is designed for humans: it's localized, it tries to prompt the user with helpful information depending on their request, if it detects that something unusual is happening it may give different output in an attempt to make this clear to the user, and so on. As a result, it is not particularly suitable for programs to parse.

Rather than trying to design output to work for both humans and computers, and serving neither audience well, we elected to create a separate interface to make programmatically extracting information from monotone easier. The command line interface has a command `automate`; this command has subcommands that print various sorts of information on standard output, in simple, consistent, and easily parseable form.

For details of this interface, see Section 5.9 [Automation], page 88.

3.4 Inodeprints

Fairly often, in order to accomplish its job, monotone has to look at your working copy and figure out what has been changed in it since your last commit. Commands that do this include `status`, `diff`, `update`, `commit`, and others. There are two different techniques it can use to do this. The default, which is sufficient for most projects, is to simply read every file in the working copy, compute their SHA1 hash, and compare them to the hashes monotone has stored. This is very safe and reliable, and turns out to be fast enough for most projects. However, on very large projects, ones whose source trees are many megabytes in size, it can become unacceptably slow.

The other technique, known as *inodeprints*, is designed for this situation. When running in inodeprints mode, monotone does not read the whole working copy; rather, it keeps a cache of interesting information about each file (its size, its last modification time, and so on), and skips reading any file for which these values have not changed. This is inherently somewhat less safe, and, as mentioned above, unnecessary for most projects, so it is disabled by default.

If you do determine that it is necessary to use inodeprints with your project, it is simple to enable them. Simply run `monotone refresh_inodeprints`; this will enable inodeprints mode and generate an initial cache. If you ever wish to turn them off again, simply delete the file ‘`MT/inodeprints`’. You can at any time delete or truncate the ‘`MT/inodeprints`’ file; monotone uses it only as a cache and will continue to operate correctly.

Normally, instead of enabling this up on a per-working-copy basis, you will want to simply define the `use_inodeprints` hook to return `true`; this will automatically enable inodeprints mode in any new working copies you create. See Chapter 6 [Hook Reference], page 105 for details.

3.5 Quality Assurance

Monotone was constructed to serve both as a version control tool and as a quality assurance tool. The quality assurance features permit users to ignore, or "filter out", versions which do not meet their criteria for quality. This section describes the way monotone represents and reasons about quality information.

Monotone often views the collection of revisions as a directed graph, in which revisions are the nodes and changes between revisions are the edges. We call this the *revision graph*. The revision graph has a number of important subgraphs, many of which overlap. For example, each branch is a subgraph of the revision graph, containing only the nodes carrying a particular `branch` cert.

Many of monotone's operations involve searching the revision graph for the ancestors or descendents of a particular revision, or extracting the "heads" of a subgraph, which is the subgraph's set of nodes with no descendents. For example, when you run the `update` command, monotone searches the subgraph consisting of descendents of the base revision of the current working copy, trying to locate a unique head to update the base revision to.

Monotone's quality assurance mechanisms are mostly based on restricting the subgraph each command operates on. There are two methods used to restrict the subgraph:

- By restricting the set of trusted `branch` certificates, you can require that specific code reviewers have approved of each edge in the subgraph you focus on.
- By restricting the set of trusted `testresult` certificates, you can require that the *end-points* of an update operation have a certificate asserting that the revision in question passed a certain test, or testsuite.

The evaluation of trust is done on a cert-by-cert basis by calling a set of lua hooks: `get_revision_cert_trust`, `get_manifest_cert_trust` and `get_file_cert_trust`. These hooks are only called when a cert has at least one good signature from a known key, and are passed *all* the keys which have signed the cert, as well as the cert's ID, name and value. The hook can then evaluate the set of signers, as a group, and decide whether to grant or deny trust to the assertion made by the cert.

The evaluation of testresults is controlled by the `accept_testresult_change` hook. This hook is called when selecting update candidates, and is passed a pair of tables describing the `testresult` certs present on the source and proposed destination of an update. Only if the change in test results are deemed "acceptable" does monotone actually select an update target to merge into your working copy.

For details on these hooks, see the Chapter 6 [Hook Reference], page 105.

3.6 Vars

Every monotone database has a set of *vars* associated with it. Vars are simple configuration variables that monotone refers to in some circumstances; they are used for configuration that monotone needs to be able to modify itself, and that should be per-database (rather than per-user or per-working copy, both of which are supported by '`monotonerc`' scripts). Vars are local to a database, and never transferred by netsync.

A var is a *name* = *value* pairing inside a *domain*. Domains define what the vars inside it are used for; for instance, one domain might contain database-global settings, and particular vars inside it would define things like that database's default netsync server. Another domain might contain key fingerprints for servers that monotone has interacted with in the past, to detect man-in-the-middle attacks; the vars inside this domain would map server names to their fingerprints.

You can set vars with the `set` command, delete them with the `unset` command, and see them with the `ls vars` command. See the documentation for these specific commands for more details.

Existing vars

There are several pre-defined domains that monotone knows about:

`database` Contains database-global configuration information. Defined names are:

`default-server`
The default server for netsync operations to use. Automatically set by first use of netsync.

`default-include-pattern`
The default branch glob pattern for netsync operations to use. Automatically set by first use of netsync.

`known-servers`
Contains key hashes for servers that we have netsynced with in the past. Analogous to `ssh`'s '`known_hosts`' file, this is needed to detect man-in-the-middle attacks. Automatically set the first time you netsync with any given server. If that server's key later changes, monotone will notice, and refuse to connect until you have run `monotone unset known-servers` *`server-name`*.

3.7 Reserved Files

A monotone working copy consists of control files and non-control files. Each type of file can be versioned or non-versioned. These classifications lead to four groups of files:

- versioned control files
- non-versioned control files
- versioned non-control files
- non-versioned non-control files

Control files contain special content formatted for use by monotone. Versioned files are recorded in a monotone database and have their state tracked as they are modified.

If a control file is versioned, it is considered *part of* the state of the working copy, and will be recorded as a manifest entry. If a control file is not versioned, it is used to *manage* the state of the working copy, but it not considered an intrinsic part of it.

Most files you manage with monotone will be versioned non-control files. For example, if you keep source code or documents in a monotone database, they are versioned non-control files. Non-versioned, non-control files in your working copy are generally temporary or junk files, such as backups made by editors or object files made by compilers. Such files are ignored by monotone.

Identifying control files

Control files are identified by their names. Non-control files can have any name *except* the names reserved for control files. The names of control files follow a regular pattern:

Versioned control files
: Any file name beginning with '`.mt-`'

Non-versioned control files
: Any file in the directory '`MT/`'

Existing control files

The following control files are currently used. More control files may be added in the future, but they will follow the patterns given above.

'`.mt-attrs`'
: Contains versioned attributes of files, associated with the files' pathnames.

'`.mt-ignore`'
: Contains a list of regular expression patterns, one per line. If it exists, any file with a name matching one of these patterns is ignored.

'`MT/wanted-testresults`'
: Contains a list of testresult key names, one per line. If it exists, update will only select revisions that do not have regressions according to the given testresult keys.

'`MT/revision`'
: Contains the identity of the "base" revision of the working copy. Each working copy has a base revision. When the working copy is committed, the base revision is considered to be the ancestor of the committed revision.

‘`MT/options`’
: Contains “sticky” command-line options such as ‘`--db`’ or ‘`--branch`’, such that you do not need to enter them repeatedly after checking out a particular working copy.

‘`MT/work`’
: Contains a list of additions, deletions, and renames which have occurred in the current working copy, relative to the base version.

‘`MT/log`’
: Contains log messages to append to the “changelog” cert upon commit. The user may add content to this file while they work. Upon a successful commit monotone will empty the file making it ready for the next edit/commit cycle.

‘`MT/inodeprints`’
: If this file exists, monotone considers the directory to be in Section 3.4 [Inodeprints], page 50 mode, and uses this file to cache the inodeprints.

‘`MT/debug`’
: If monotone detects a bug in itself or crashes, then before exiting it dumps a log of its recent activity to this file, to aid in debugging.

3.8 Reserved Certs

Every certificate has a name. Some names have meaning which is built in to monotone, others may be used for customization by a particular user, site, or community. If you wish to define custom certificates, you should prefix such certificate names with `x-`. For example, if you want to make a certificate describing the existence of security vulnerabilities in a revision, you might wish to create a certificate called `x-vulnerability`. Monotone reserves all names which do not begin with `x-` for possible internal use. If an `x-` certificate becomes widely used, monotone will likely adopt it as a reserved cert name and standardize its semantics.

Most reserved certificate names have no meaning yet; some do. Usually monotone is also responsible for *generating* these certificates, so you should generally have no cause to make them yourself. They are described here to help you understand monotone's operation.

The pre-defined, reserved certificate names are:

`author`
: This cert's value is the name of a person who committed the revision the cert is attached to. The cert is generated when you commit a revision. It is displayed by the `log` command.

`branch`
: This cert's value is the name of a branch. A `branch` cert associates a revision with a branch. The revision is said to be "in the branch" named by the cert. The cert is generated when you commit a revision, either directly with the `commit` command or indirectly with the `merge` or `propagate` commands. The `branch` certs are read and directly interpreted by *many* monotone commands, and play a fundamental role in organizing work in any monotone database.

`changelog`
: This cert's value is the change log message you provide when you commit a revision. It is displayed by the `log` command.

`comment`
: This cert's value is an additional comment, usually provided after committing, about a revision. Certs with the name `comment` can be applied to files as well, and will be shown by the `log` command.

`date`
: This cert's value is an ISO date string indicating the time at which a revision was committed. It is displayed by the `log` command, and may be used as an additional heuristic or selection criterion in other commands in the future.

`tag`
: This cert's value is a symbolic name given to a revision, which may be used in the future as a way of selecting versions for `checkout`.

`testresult`
: This cert's value is interpreted as a boolean string, either `0` or `1`. It is generated by the `testresult` command and represents the results of running a particular test on the underlying revision. Typically you will make a separate signing key for each test you intend to run on revisions. This cert influences the `update` algorithm.

3.9 Naming Conventions

Some names in monotone are private to your work, such as filenames. Other names are potentially visible outside your project, such as RSA key identifiers or branch names. It is possible that if you choose such names carelessly, you will choose a name which someone else in the world is using, and subsequently you may cause confusion when your work and theirs is received simultaneously by some third party.

We therefore recommend two naming conventions:

- For RSA keys, use the name of an active email address you own. This will minimize conflicts, and also serves as a mnemonic to associate your personal *identity* with signatures made with your key. For example, monotone's primary author uses the key identifier `graydon@pobox.com`.
- For branch names, select any name you like but prefix it with the "inverted domain name" of a DNS domain you control or are otherwise authorized to use. This behavior mimics the package naming convention in the java programming language. For example, monotone itself is developed within the `net.venge.monotone` branch, because the author owns the DNS domain `venge.net`.

3.10 File Attributes

Monotone contains a mechanism for storing *persistent file attributes*. These differ from file certificates in an important way: attributes are associated with a path name in your working copy, rather than a particular version of a file. Otherwise they are similar: a file attribute associates a simple name/value pair with a file in your working copy.

The attribute mechanism is motivated by the fact that some people like to store executable programs in version control systems, and would like the programs to remain executable when they check out a working copy. For example, the `configure` shell script commonly shipped with many programs should be executable.

Similarly, some people would like to store devices, symbolic links, read-only files, and all manner of extra attributes of a file, not directly related to a file's data content.

Rather than try to extend the manifest file format to accommodate attributes, monotone requires that you place your attributes in a specially named file in the root of your working copy. The file is called '`.mt-attrs`', and it has a simple stanza-based format, for example:

```
file "analyze_coverage"
execute "true"

file "autogen.sh"
execute "true"
otherattr "bob"
```

Each stanze of the '`.mt-attrs`' file assigns attributes to a file in your working copy. The first line of each stanza is `file` followed by the quoted name of the file you want to assign attributes to. Each subsequent line is the name of an attribute, followed by a quoted value for that attribute. Stanzas are separated by blank lines.

As a convenience, you can use the `monotone attr` command to set and view the values of these attributes; see Section 5.2 [Working Copy], page 69.

You can tell monotone to automatically take actions based on these attributes by defining hooks; see the `attr_functions` entry in Chapter 6 [Hook Reference], page 105.

Every time your working copy is written to, monotone will look for the '`.mt-attrs`' file, and if it exists, run the corresponding hooks registered for each attribute found in the file. This way, you can extend the vocabulary of attributes understood by monotone simply by writing new hooks.

Aside from its special interpretation, the '`.mt-attrs`' file is a normal text file. If you want other people to see your attributes, you should `add` and `commit` the '`.mt-attrs`' file in your working copy. If you make changes to it which conflict with changes other people make, you will have to resolve those conflicts, as plain text, just as with any other text file in your working copy.

3.11 Merging

Monotone has two merging modes, controlled by the `manual_merge` attribute. By default all files are merged in automatic mode, unless the `manual_merge` attribute for that file is present and `true`. In automatic mode files are merged without user intervention, using monotone internal three-way merging algorithm. Only if there are conflicts or an ancestor is not available monotone switches to manual mode, essentially escalating the merging to the user. When working in manual mode, monotone invokes the merge2 (for two-way merging) or merge3 (three-way) hooks to start an user defined external merge tool. If the tool terminates without writing the merged file, monotone aborts the merging, reverting any changes made. By redefining the aforementioned hooks the user can not only choose a preferred merge tool, but even select different programs for different file types. For example, gimp for .png files, OpenOffice.org for .doc, and so on. Starting with monotone 0.20, the `manual_merge` attribute is automatically set at add time for all “binary” files, i.e. all files for wich the `binary_file` hook returns true. Currently, this means all files with extension gif, jpeg, png, bz2, gz and zip, plus files containing at least one of the following bytes:

```
0x00 thru 0x06
0x0E thru 0x1a
0x1c thru 0x1f
```

The attribute could also be manually forced or removed using the appropriate monotone commands. Remember that monotone switches to manual merging even if only one of the files to be merged has the `manual_merge` attribute set.

3.12 Migrating and Dumping

While the state of your database is logically captured in terms of a packet stream, it is sometimes necessary or desirable (especially while monotone is still in active development) to modify the SQL table layout or storage parameters of your version database, or to make backup copies of your database in plain text. These issues are not properly addressed by generating packet streams: instead, you must use *migration* or *dumping* commands.

The `monotone db migrate` command is used to alter the SQL schema of a database. The schema of a monotone database is identified by a special hash of its generating SQL, which is stored in the database's auxiliary tables. Each version of monotone knows which schema version it is able to work with, and it will refuse to operate on databases with different schemas. When you run the `migrate` command, monotone looks in an internal list of SQL logic which can be used to perform in-place upgrades. It applies entries from this list, in order, attempting to change the database it *has* into the database it *wants*. Each step of this migration is checked to ensure no errors occurred and the resulting schema hashes to the intended value. The migration is attempted inside a transaction, so if it fails — for example if the result of migration hashes to an unexpected value — the migration is aborted.

If more drastic changes to the underlying database are made, such as changing the page size of sqlite, or if you simply want to keep a plain text version of your database on hand, the `monotone db dump` command can produce a plain ASCII SQL statement which generates the state of your database. This dump can later be reloaded using the `monotone db load` command.

Note that when reloading a dumped database, the schema of the dumped database is *included* in the dump, so you should not try to `init` your database before a `load`.

3.13 Importing from CVS

Monotone is capable of reading CVS files directly and importing them into a database. This feature is still somewhat immature, but moderately large "real world" CVS trees on the order of 1GB have successfully been imported.

Note however that the machine requirements for CVS trees of this size are not trivial: it can take several hours on a modern system to reconstruct the history of such a tree and calculate the millions of cryptographic certificates involved. We recommend experimenting with smaller trees first, to get a feel for the import process.

We will assume certain values for this example which will differ in your case:

- Your domain name, `example.net` in this example.
- Your key name, `import@example.net` in this example.
- Your project name, `wobbler` in this example.
- Your database name, '`test.db`' in this example.
- Your CVS repository path, '`/usr/local/cvsroot`' in this example.
- The CVS module name for your project, `wobbler` in this example.

Accounting for these differences at your site, the following is an example procedure for importing a CVS repository "from scratch", and checking the resulting head version of the import out into a working copy:

```
$ monotone --db=test.db db init
$ monotone --db=test.db genkey import@example.net
$ monotone --db=test.db --branch=net.example.wobbler cvs_import /usr/local/cvsroot/wobbler
$ monotone --db=test.db --branch=net.example.wobbler checkout wobber-checkout
```

4 CVS Phrasebook

This chapter translates common CVS commands into monotone commands. It is an easy alternative to reading through the complete command reference.

Checking Out a Tree

```
$ CVSROOT=:pserver:cvs.foo.com/wobbler $ monotone pull www.foo.com com.foo.wobbler*
$ cvs -d $CVSROOT checkout -r 1.2      $ monotone checkout --revision=fe37 wobbler
```

The CVS command contacts a network server, retrieves a revision, and stores it in your working copy. There are two cosmetic differences with the monotone command: remote databases are specified by hostnames and globs, and revisions are denoted by SHA1 values (or selectors).

There is also one deep difference: pulling revisions into your database is a separate step from checking out a single revision; after you have pulled from a network server, your database will contain *several* revisions, possibly the entire history of a project. Checking out is a separate step, after communication, which only copies a particular revision out of your database and into a named directory.

Committing Changes

```
$ cvs commit -m "log message"          $ monotone commit --message="log message"
                                       $ monotone push www.foo.com com.foo.wobbler*
```

As with other networking commands, the communication step with monotone is explicit: committing changes only saves them to the local database. A separate command, `push`, sends the changes to a remote database.

Incorporating New Changes

```
$ cvs update -d                        $ monotone pull www.foo.com com.foo.wobbler*
                                       $ monotone merge
                                       $ monotone update
```

This command, like other networking commands, involves a separate communication step with monotone. The extra command, `merge`, ensures that the branch your are working on has a unique head. You can omit the `merge` step if you only want `update` to examine descendents of your base revision, and ignore other heads on your branch.

Moving Working Copy to Another Revision

```
$ cvs update -r FOO_TAG -d             $ monotone update -r 830ac1a5f033825ab364f911608ec294fe37f7bc
                                       $ monotone update -r t:FOO_TAG
```

With a revision parameter, the `update` command operates similarly in monotone and CVS. One difference is that a subsequent `commit` will be based off the chosen revision in monotone, while a `commit` in the CVS case is not possible without going back to the branch head again. This version of `update` can thus be very useful if, for example, you discover that the tree you are working against is somehow broken — you can `update` to an older non-broken version, and continue to work normally while waiting for the tree to be fixed.

Viewing Differences

```
$ cvs diff                          $ monotone diff

$ cvs diff -r 1.2 -r 1.4 myfile     $ monotone diff -r 3e7db -r 278df myfile
```

Monotone's `diff` command is modeled on that of CVS, so the main features are the same: `diff` alone prints the differences between your working copy and its base revision, whereas `diff` accompanied by two revision numbers prints the difference between those two revisions. The major difference between CVS and monotone here is that monotone's revision numbers are *revision IDs*, rather than file IDs. If one leaves off the file argument, then diff can print the difference between two entire trees.

Showing Working Copy Status

```
$ cvs status                        $ monotone status
```

This command operates similarly in monotone and CVS. The only major difference is that monotone's `status` command always gives a status of the whole tree, and outputs a more compact summary than CVS.

Adding Directories and Files to Working Copy

```
$ cvs add dir                       $ monotone add dir/subdir/file.txt
$ cvs add dir/subdir
$ cvs add dir/subdir/file.txt
```

Monotone does not explicitly store directories, so adding a file only involves adding the file's complete path, including any directories. Directories are created as needed, and empty directories are ignored.

Removing Directories and Files from Working Copy

```
$ rm file.txt                       $ monotone drop file.txt
$ cvs remove file.txt
```

Monotone does not require that you erase a file from the working copy before you drop it. Dropping a file merely removes its entry in the manifest of the current revision.

Viewing History

```
$ cvs log [file]                    $ monotone log [file]
```

Unlike CVS log, monotone log can also be used without a working directory; but in this case you must pass a '`--revision`' argument to tell monotone where to start displaying the log from.

Importing a New Project

```
$ cvs import wobbler vendor start   $ monotone --db=/path/to/database.db -
                                    -branch=com.foo.wobbler setup .
                                    $ monotone add .
                                    $ monotone commit
```

The `setup` command turns an ordinary directory into a monotone working copy. After that, you can add your files and commit them as usual.

Initializing a Repository

```
$ cvs init -d /path/to/repository      $ monotone db init --db=/path/to/database.db
```

Monotone's "repository" is a single-file database, which is created and initialized by this command. This file is only ever used by you, and does not need to be in any special location, or readable by other users.

5 Command Reference

Monotone has a large number of commands. To help navigate through them all, commands are grouped into logical categories.

5.1 Tree

`monotone cat` *`path`*
`monotone cat --revision=`*`id path`*

Write the contents of a specific file *path* to standard output.

Without a '`--revision`' argument, the command outputs the contents of *path* as found in the current revision. This requires the command be executed from within a working copy.

With an explicit '`--revision`' argument, the command outputs contents of *path* at that revision.

`monotone checkout --revision=`*`id directory`*
`monotone co --revision=`*`id directory`*
`monotone --branch=`*`branchname`*` checkout `*`directory`*
`monotone --branch=`*`branchname`*` co `*`directory`*

These commands copy a revision *id* out of your database, writing the string *id* into the file '*`directory`*`/MT/revision`'. These commands then copy every file version listed in the revision's manifest to paths under *directory*. For example, if the revision's manifest contains these entries:

```
84e2c30a2571bd627918deee1e6613d34e64a29e  Makefile
c61af2e67eb9b81e46357bb3c409a9a53a7cdfc6  include/hello.h
97dfc6fd4f486df95868d85b4b81197014ae2a84  src/hello.c
```

Then the following files are created:

```
directory/Makefile
directory/include/hello.h
directory/src/hello.c
```

If you wish to `checkout` in the current directory, you can supply the special name '`.`' (a single period) for *directory*.

If no *id* is provided, as in the latter two commands, you *must* provide a *branchname*; monotone will attempt to infer *id* as the unique head of *branchname* if it exists.

`monotone disapprove` *`id`*

This command records a disapproval of the changes between *id*'s ancestor and *id*. It does this by committing the *inverse* changes as a new revision descending from *id*. The new revision will show up as a new head and thus a subsequent `merge` will incorporate the inverse of the disapproved changes in the other head(s).

Note that this command only works if *id* has exactly one ancestor.

`monotone heads --branch=`*`branchname`*

This command lists the "heads" of *branchname*.

The "heads" of a branch is the set of revisions which are members of the branch, but which have no descendents. These revisions are generally the "newest" revisions committed by you or your colleagues, at least in terms of ancestry. The heads of a branch may not be the newest revisions, in terms of time, but synchronization of computer clocks is not reliable, so monotone usually ignores time.

`monotone merge [--branch=`*`branchname`*`]`
`monotone merge --lca [--branch=`*`branchname`*`]`

This command merges the "heads" of *branchname*, if there are multiple heads, and commits the results to the database, marking the resulting merged revision as a member of *branchname*. The merged revision will contain each of the head revision IDs as ancestors.

Merging is performed by repeated pairwise merges: two heads are selected, then their least common ancestor is located in the ancestry graph and these 3 revisions are provided to the built-in 3-way merge algorithm. The process then repeats for each additional head, using the result of each previous merge as an input to the next.

The '`--lca`' option is provided as a temporary workaround for some inherent problems with 3-way merging; if you encounter a huge number of spurious conflicts, or get an invariant failure, then you may wish to try using '`--lca`'. Using it *can*, in certain circumstances, cause some edits that should be conflicts to merge cleanly instead, which is why it is not the default; but these cases are rare, and until we can find something better than 3-way merging to use, you may find this option more convenient than `explicit_merge`

`monotone propagate `*`sourcebranch destbranch`*
`monotone propagate --lca `*`sourcebranch destbranch`*

This command takes a unique head from *sourcebranch* and merges it with a unique head of *destbranch*, using the least common ancestor of the two heads for a 3-way merge. The resulting revision is committed to *destbranch*. If either *sourcebranch* or *destbranch* has multiple heads, `propagate` aborts, doing nothing.

The purpose of `propagate` is to copy all the changes on *sourcebranch*, since the last `propagate`, to *destbranch*. This command supports the idea of making separate branches for medium-length development activities, such as maintenance branches for stable software releases, trivial bug fix branches, public contribution branches, or branches devoted to the development of a single module within a larger project.

The '`--lca`' option is provided as a temporary workaround for some inherent problems with 3-way merging; if you encounter a huge number of spurious conflicts, or get an invariant failure, then you may wish to try using '`--lca`'. Using it *can*, in certain circumstances, cause some edits that should be conflicts to merge cleanly instead, which is why it is not the default; but these cases are rare, and until we can find something better than 3-way merging to use, you may find this option more convenient than `explicit_merge`

`monotone explicit_merge `*`id id destbranch`*
`monotone explicit_merge `*`id id ancestor destbranch`*

This command merges exactly the two *id*s you give it, and places the result in branch *destbranch*. It is useful when you need more control over the merging process than `propagate` or `merge` give you. For instance, if you have a branch with three heads, and you only want to merge two of them, you can use this command. Or if you have a branch with two heads, and you want to propagate one of them to another branch, again, you can use this command. If the optional *ancestor* argument is given, the merge uses that revision as the common ancestor instead of the default ancestor.

5.2 Working Copy

`monotone add` *`pathname...`*

This command places “add” entries for the paths specified in *pathname...* in the working copy’s “work list”. The work list of your working copy is located at ‘`MT/work`’, and is a list of explicit pathname changes you wish to commit at some future time, such as addition, removal or renaming of files.

While this command places an “add” entry on your work list, it does not immediately affect your database. When you `commit` your working copy, monotone will use the work list to build a new revision, which it will then commit to the database. The new revision will have any added entries inserted in its manifest.

`monotone drop` *`pathname...`*

This command places “drop” entries for the paths specified in *pathname...* in the working copy’s “work list”. The work list of your working copy is located at ‘`MT/work`’, and is a list of explicit pathname changes you wish to commit at some future time, such as addition, removal, or renaming of files. This command also removes any attributes on *pathname*; see Section 3.10 [File Attributes], page 57 for more details.

While this command places a “drop” entry on your work list, it does not immediately affect your database. When you `commit` your working copy, monotone will use the work list to build a new revision, which it will then commit to the database. The new revision will have any dropped entries removed from its manifest.

Currently this command does *not* actually delete the file *src* in your filesystem; if you want to actually delete the file, you should run `drop`, and then perform the actual delete using whatever mechanism you normally use to delete files.

`monotone rename` *`src dst`*

This command places “rename” entries for the paths specified in *src* and *dst* in the working copy’s “work list”. The work list of your working copy is located at ‘`MT/work`’, and is a list of explicit pathname changes you wish to commit at some future time, such as addition, removal, or renaming of files. This command also moves any attributes on *src* to *dst*; see Section 3.10 [File Attributes], page 57 for more details.

While this command places a “rename” entry on your work list, it does not immediately affect your database. When you `commit` your working copy, monotone will use the work list to build a new revision, which it will then commit to the database. The new revision will have any renamed entries in its manifest adjusted to their new names.

Currently this command does *not* actually rename the file *src* in your filesystem; after you run this command, you should do the actual rename, using whatever mechanism you normally use to rename files.

`monotone commit`
`monotone commit --message=`*`logmsg`*
`monotone commit --message-file=`*`logfile`*
`monotone commit` *`pathname...`*
`monotone commit --message=`*`logmsg pathname...`*
`monotone commit --message-file=`*`logfile pathname...`*

This command looks at your working copy, decides which files have changed, and saves the changes to your database. It does this by loading the revision named in the '`MT/revision`' file, locating the base manifest for your working copy, applying any changes described in the '`MT/work`' file, and then comparing the updated base manifest to the files it finds in your working copy, to determine which files have been edited.

For each edited file, a delta is copied into the database. Then the newly constructed manifest is recorded (as a delta) and finally the new revision. Once all these objects are recorded in you database, `commit` overwrites the '`MT/revision`' file with the new revision ID, and deletes the '`MT/work`' file.

Specifying pathnames to `commit` restricts the set of changes that are visible and results in only a partial commit of the working copy. Changes to files not included in the specified set of pathnames will be ignored and will remain in the working copy until they are included in a future commit. With a partial commit, only the relevant entries in the '`MT/work`' file will be removed and other entries will remain for future commits.

From within a subdirectory of the working copy the `commit` command will, by default, include *all changes* in the working copy. Specifying only the pathname `"."` will restrict `commit` to files changed within the current subdirectory of the working copy.

The '`--message`' and '`--message-file`' options are mutually exclusive. Both provide a *logmsg* describing the commit. '`--message-file`' actually specifies the name of the file containing the log message, while '`--message`' provides it directly.

The '`MT/log`' file can be edited by the user during their daily work to record the changes made to the working copy. When running the `commit` command without a *logmsg* supplied, the contents of the '`MT/log`' file will be read and passed to the Lua hook `edit_comment` as a second parameter named *user_log_content*. If the commit is successful, the '`MT/log`' file is cleared of all content making it ready for another edit/commit cycle.

If a '`--branch`' option is specified, the `commit` command commits to this branch (creating it if necessary). The branch becomes the new default branch of the working copy.

The `commit` command also synthesizes a number of certificates, which it attaches to the new manifest version and copies into your database:

- An `author` cert, indicating the person responsible for the changes leading to the new revision. Normally this defaults to your signing key or the return value of the `get_author` hook; you may override this by passing the '`--author`' option to commit. This is useful when committing a patch on behalf of someone else, or when importing "by hand" from another version control system.

- A `branch` cert, indicating the branch the committed revision belongs to.
- A `date` cert, indicating when the new revision was created. Normally this defaults to the current time; you may override this by passing the '`--date`' option to commit. This is useful when importing "by hand" from another version control system.
- A `changelog` cert, containing the "log message" for these changes. If you provided *logmsg* on the command line, this text will be used, otherwise `commit` will run the Lua hook `edit_comment (`*`commentary, user_log_content`*`)`, which typically invokes an external editor program, in which you can compose and/or review your log message for the change.

`monotone revert`
`monotone revert` *`pathname...`*

With no files given, this command changes your working copy, so that changes you have made since the last checkout or update are discarded. It does this by changing every file listed in the working copy's base manifest to have contents equal to the SHA1 value listed in the manifest, and by erasing the '`MT/work`' file.

If files or directories are given as arguments, only those files and directories are affected instead of the entirety of your working copy.

From within a subdirectory of the working copy the `revert` command will, by default, revert *all changes* in the working copy. Specifying only the pathname `"."` will restrict `revert` to files changed within the current subdirectory of the working copy. *Caution should be used when reverting files to ensure that the correct set of files is reverted.*

`monotone update`
`monotone update --revision=`*`revision`*

Without a '`--revision`' argument, this command incorporates "recent" changes found in your database into your working copy. It does this by performing 3 separate stages. If any of these stages fails, the update aborts, doing nothing. The stages are:

- Examine the ancestry graph of revisions in your database, and (subject to trust evaluation) select the set of all descendents of your working copy's base revision. Call this set the "candidates" of the update.
- Remove any candidates which lack acceptable testresult certificates. From the remaining candidates, select the deepest child by ancestry and call it the "target" of the update.
- Merge the target of the update with the working copy, in memory, and if the merge is successful, write the result over top of the working copy.

With an explicit '`--revision`' argument, the command uses that revision as the update target instead of finding an acceptable candidate.

The effect is always to take whatever changes you have made in the working copy, and to "transpose" them onto a new revision, using monotone's

3-way merge algorithm to achieve good results. Note that with the explicit '`--revision`' argument, it is possible to update "backwards" or "sideways" in history — for example, reverting to an earlier revision, or if your branch has two heads, updating to the other. In all cases, the end result will be whatever revision you specified, with your local changes (and only your local changes) applied.

If a '`--branch`' option is specified, the `update` command tries to select the revision to update to from this branch. The branch becomes the new default branch of the working copy (even if you also specify an explicit '`--revision`' argument).

`monotone refresh_inodeprints`

This command puts the current working copy into Section 3.4 [Inodeprints], page 50 mode, if it was not already, and forces a full inodeprints cache refresh. After running this command, you are guaranteed that your working copy is in inodeprints mode, and that the inodeprints cache is accurate and up to date.

5.3 Network

`monotone serve` *`address`*`[:`*`port`*`]` *`glob`* `[...] [--exclude=`*`exclude-glob`*`]`
`monotone pull [--set-default] [`*`address`*`[:`*`port`*`] [`*`glob`*` [...]`
`[--exclude=`*`exclude-glob`*`]]]`
`monotone push [--set-default] [`*`address`*`[:`*`port`*`] [`*`glob`*` [...]`
`[--exclude=`*`exclude-glob`*`]]]`
`monotone sync [--set-default] [`*`address`*`[:`*`port`*`] [`*`glob`*` [...]`
`[--exclude=`*`exclude-glob`*`]]]`

These commands operate the "netsync" protocol built into monotone. This is a custom protocol for rapidly synchronizing two monotone databases using a hash tree index. The protocol is "peer to peer", but requires one peer to listen for incoming connections (the server) and the other peer (the client) to connect to the server.

The network *address* specified in each case should be the same: a host name to listen on, or connect to, optionally followed by a colon and a port number. The default port number is 5253. The *glob* parameters indicate a set of branches to exchange. Multiple *glob* and '`--exclude`' options can be specified; every branch which matches a *glob* exactly, and does not match an *exclude-glob*, will be indexed and made available for synchronization.

For example, perhaps Bob and Alice wish to synchronize their `net.venge.monotone.win32` and `net.venge.monotone.i18n` branches. Supposing Alice's computer has hostname `alice.someisp.com`, then Alice might run:

```
$ monotone serve alice.someisp.com "net.venge.monotone*"
```

And Bob might run

```
$ monotone sync alice.someisp.com "net.venge.monotone*"
```

When the operation completes, all branches matching `net.venge.monotone*` will be synchronized between Alice and Bob's databases.

The `pull`, `push`, and `sync` commands only require you pass *address* and *glob* the first time you use one of them; monotone will memorize this use and in the future default to the same server and glob. For instance, if Bob wants to `sync` with Alice again, he can simply run:

```
$ monotone sync
```

Of course, he can still `sync` with other people and other branches by passing an address or address plus globs on the command line; this will not affect his default affinity for Alice. If you ever do want to change your defaults, simply pass the '`--set-default`' option when connecting to the server and branch pattern that you want to make the new default.

In the server, different permissions can be applied to each branch; see the hooks `get_netsync_read_permitted` and `get_netsync_write_permitted` (see Chapter 6 [Hook Reference], page 105).

If a '`--pid-file`' option is specified, the command `serve` will create the specified file and record the process identifier of the server in the file. This file can then be read to identify specific monotone server processes.

The syntax for patterns is very simple. `*` matches 0 or more arbitrary characters. `?` matches exactly 1 arbitrary character. `{foo,bar,baz}` matches "foo", or "bar", or "baz". These can be combined arbitrarily. A backslash, `\`, can be prefixed to any character, to match exactly that character — this might be useful in case someone, for some odd reason, decides to put a "*" into their branch name.

5.4 Informative

`monotone status`
`monotone status pathname...`

This command prints a description of the "status" of your working copy. In particular, it prints:

- The "base revision ID" and "base manifest ID", which are referenced by the file '`MT/revision`', and which your working copy is an in-progress descendent of.
- The "current manifest ID", which is the ID of the manifest which results from applying '`MT/work`' to the base manifest, and updating any SHA1 values of files to reflect changes you have made to the working copy. In other words, the current manifest ID is the ID which would accompany any revision you would commit, if you ran `monotone commit`.
- A list of logical changes between the base and current manifest versions, such as adds, drops, renames, and patches.

Specifying optional *pathname...* arguments to the `status` command restricts the set of changes that are visible and results in only a partial status of the working copy. Changes to files not included in the specified set of pathnames will be ignored.

From within a subdirectory of the working copy the `status` command will, by default, include *all changes* in the working copy. Specifying only the pathname `"."` will restrict `status` to files changed within the current subdirectory of the working copy.

`monotone log`
`monotone log [--last=n] [--revision=id [...]] [--brief] [file [...]]`

This command prints out a log, in reverse-ancestry order, of small history summaries. Each summary contains author, date, changelog and comment information associated with a revision. If `--brief` is given, the output consists of one line per revision with the revision ID, the author, the date and the branches (separated with commas).

If `--last=n` is given, at most that many log entries will be given.

If one or more revision IDs are given, the command starts tracing back through history from these revisions, otherwise it starts from the base revision of your working copy.

If one or more files are given, the command will only log the revisions where those files are changed.

`monotone annotate file`
`monotone annotate --revision=id file`

Dumps an annotated copy of the file to stdout. Each line of the file is translated to <revision id>: <line> in the output, where <revision id> is the revision in which that line of the file was last edited.

`monotone complete file` *`partial-id`*
`monotone complete [--brief] key` *`partial-id`*
`monotone complete manifest` *`partial-id`*
`monotone complete [--brief] revision` *`partial-id`*

These commands print out all known completions of a partial SHA1 value, listing completions which are `file`, `manifest` or `revision` IDs depending on which variant is used. For example, suppose you enter this command and get this result:

```
$ monotone complete manifest fa36
fa36deead87811b0e15208da2853c39d2f6ebe90
fa36b76dd0139177b28b379fe1d56b22342e5306
fa36965ec190bee14c5afcac235f1b8e2239bb2a
```

Then monotone is telling you that there are 3 manifests it knows about, in its database, which begin with the 4 hex digits `fa36`. This command is intended to be used by programmable completion systems, such as those in `bash` and `zsh`.

The complete command for keys and revisions have a `--verbose` option. Programmable completion systems can use `--verbose` output to present users with additional information about each completion option.

For example, verbose output for `revision` looks like this:

```
$ mt complete revision 01f
01f5da490941bee1f0000f0561fc62eabfb2fa23 graydon@dub.net 2003-12-03T03:14:35
01f992577bd8bcdcade0f89e724fd5dc2d2bbe8a kinetik@orcon.nz 2005-05-11T05:19:29
01faad191d8d0474777c70b4d606782942333a78 kinetik@orcon.nz 2005-04-11T04:24:01
```

`monotone diff`
`monotone diff --context`
`monotone diff --external [--diff-args=`*`argstring`*`]`
`monotone diff` *`pathname...`*
`monotone diff --revision=`*`id`*
`monotone diff --revision=`*`id pathname...`*
`monotone diff --revision=`*`id1`* `--revision=`*`id2`*
`monotone diff --revision=`*`id1`* `--revision=`*`id2 pathname...`*

These commands print out GNU "unified diff format" textual difference listings between various manifest versions. With no '`--revision`' options, `diff` will print the differences between the base revision and the current revision in the working copy.

With one '`--revision`' option, `diff` will print the differences between the revision *id* and the current revision in the working copy. With two '`--revision`' options `diff` will print the differences between revisions *id1* and *id2*, ignoring any working copy.

In all cases, monotone will print a textual summary – identical to the summary presented by `monotone status` – of the logical differences between revisions in lines proceeding the diff. These lines begin with a single hash mark `#`, and should be ignored by a program processing the diff, such as `patch`.

Specifying pathnames to the `diff` command restricts the set of changes that are visible and results in only a partial diff between two revisions. Changes to files not included in the specified set of pathnames will be ignored.

From within a subdirectory of the working copy the `diff` command will, by default, include *all changes* in the working copy. Specifying only the pathname `"."` will restrict `diff` to files changed within the current subdirectory of the working copy.

The output format of `diff` is controlled by the options '`--unified`', '`--context`', and '`--external`'. '`--unified`' requests the "unified diff" format, the default (analogous to running the program `diff -u`). '`--context`' request the "context diff" format (analogous to running the program `diff -c`). Both of these formats are generated directly by monotone, using its built-in diff algorithm.

Sometimes, you may want more flexibility in output formats; for these cases, you can use '`--external`', which causes monotone to invoke an external program to generate the actual output. By default, the external program is `diff`, and you can use the option '`--diff-args`' to pass additional arguments controlling formatting. The actual invocation of `diff`, default arguments passed to it, and so on, are controlled by the hook `external_diff`; see Section 6.1 [Hooks], page 106 for more details.

`monotone list certs` *`id`*

These commands will print out a list of certificates associated with a particular revision *id*. Each line of the print out will indicate:

- Whether the signature on the certificate is `ok` or `bad`
- The key ID of the signer of the certificate
- The name of the certificate
- The value of the certificate

For example, this command lists the certificates associated with a particular version of monotone itself, in the monotone development branch:

```
$ monotone list certs 4a96
monotone: expandeding partial id '4a96'
monotone: expanded to '4a96a230293456baa9c6e7167cafb3c5b52a8e7f'
-----------------------------------------------------------------
Key   : graydon@pobox.com
Sig   : ok
Name  : author
Value : graydon@dub.venge.net
-----------------------------------------------------------------
Key   : graydon@pobox.com
Sig   : ok
Name  : branch
Value : monotone
-----------------------------------------------------------------
Key   : graydon@pobox.com
Sig   : ok
Name  : date
Value : 2003-10-17T03:20:27
-----------------------------------------------------------------
Key   : graydon@pobox.com
Sig   : ok
Name  : changelog
Value : 2003-10-16  graydon hoare  <graydon@pobox.com>
      :
      :         * sanity.hh: Add a const version of idx().
      :         * diff_patch.cc: Change to using idx() everywhere.
      :         * cert.cc (find_common_ancestor): Rewrite to recursive
      :         form, stepping over historic merges.
      :         * tests/t_cross.at: New test for merging merges.
      :         * testsuite.at: Call t_cross.at.
      :
```

`monotone list keys`
`monotone list keys` *`pattern`*

These commands list RSA keys held in your current database. They do not print out any cryptographic information; they simply list the names of public and private keys you have on hand.

If *pattern* is provided, it is used as a glob to limit the keys listed. Otherwise all keys in your database are listed.

`monotone list branches`

This command lists all known branches in your database.

`monotone list tags`

This command lists all known tags in your database.

`monotone list vars`
`monotone list vars` *`domain`*

This command lists all vars in your database, or all vars within a given *domain*. See Section 3.6 [Vars], page 52 for more information.

`monotone list known`
`monotone list known` *`pathname...`*

This command lists all files which would become part of the manifest of the next revision if you comitted your working copy at this point.

Specifying pathnames to the `list known` command restricts the set of paths that are searched for manifest files. Files not included in the specified set of pathnames will not be listed.

From within a subdirectory of the working copy the `list known` command will, by default, search the entire working copy. Specifying only the pathname "." will restrict the search for known files to the current subdirectory of the working copy.

`monotone list unknown`
`monotone list unknown` *`pathname...`*

This command lists all files in your working copy that monotone is either ignoring or knows nothing about.

Specifying pathnames to the `list unknown` command restricts the set of paths that are searched for unknown files. Unknown files not included in the specified set of pathnames will not be listed.

From within a subdirectory of the working copy the `list unknown` command will, by default, search the entire working copy. Specifying only the pathname "." will restrict the search for unknown files to the current subdirectory of the working copy.

`monotone list ignored`
`monotone list ignored` *`pathname...`*

This command lists all files in your working copy that monotone is intentionally ignoring, due to the results of the `ignore_file (`*`filename`*`)` hook.

Specifying pathnames to the `list ignored` command restricts the set of paths that are searched for ignored files. Ignored files not included in the specified set of pathnames will not be listed.

From within a subdirectory of the working copy the `list ignored` command will, by default, search the entire working copy. Specifying only the pathname "." will restrict the search for ignored files to the current subdirectory of the working copy.

`monotone list missing`
`monotone list missing` *`pathname...`*

This command lists all files in your working copy's base manifest, which are not present in the working copy.

Specifying pathnames to the `list missing` command restricts the set of paths that are searched for missing files. Missing files not included in the specified set of pathnames will not be listed.

From within a subdirectory of the working copy the `list missing` command will, by default, search the entire working copy. Specifying only the pathname "." will restrict the search for missing files to the current subdirectory of the working copy.

5.5 Key and Cert

`monotone genkey` *`keyid`*

This command generates an RSA public/private key pair, using a system random number generator, and stores it in your database under the key name *keyid*. If the the hook `non_blocking_rng_ok()` returns `true`, the key generation will use an unlimited random number generator (such as '`/dev/urandom`'), otherwise it will use a higher quality random number generator (such as '`/dev/random`') but might run slightly slower.

The private half of the key is stored in an encrypted form, using the symmetric cipher ARC4, so that anyone accidentally reading your database cannot extract your private key and use it. You must provide a passphrase for your key when it is generated, which is used to key the ARC4 cipher. In the future you will need to enter this passphrase again each time you sign a certificate, which happens every time you `commit` to your database. You can tell monotone to automatically use a certain passphrase for a given key using the `get_passphrase(`*`keypair_id`*`)`, but this significantly increases the risk of a key compromise on your local computer. Be careful using this hook.

`monotone dropkey` *`keyid`*

This command drops the public and/or private key. If both exist, both are dropped, if only one exists, it is dropped. This command should be used with caution as changes are irreversible without a backup of the key(s) that were dropped. Note also that the private key is not guaranteed to actually be erased from your database file — if you are going to make the database file public, you should use `db dump` and `db load` to import into a fresh database.

`monotone chkeypass` *`id`*

This command lets you change the passphrase of the private half of the key *id*.

`monotone cert` *`id certname`*
`monotone cert` *`id certname certval`*

These commands create a new certificate with name *certname*, for a revision with version *id*. If *certval* is provided, it is the value of the certificate. Otherwise the certificate value is read from `stdin`.

`monotone trusted` *`id certname certval signers`*

This command lets you test your revision trust hook `get_revision_cert_trust` (see Chapter 6 [Hook Reference], page 105). You pass it a revision ID, a certificate name, a certificate value, and one or more key IDs, and it will tell you whether, under your current settings, Monotone would trust a cert on that revision with that value signed by those keys.

5.6 Certificate

`monotone approve` *`id`*

This command is a synonym for `monotone cert` *`id`* `branch` *`branchname`* where *branchname* is the current branch name (either deduced from the working copy or from the '`--branch`' option).

`monotone comment` *`id`*
`monotone comment` *`id comment`*

These commands are synonyms for `monotone cert` *`id`* `comment` *`comment`*. If *comment* is not provided, it is read from `stdin`.

`monotone tag` *`id tagname`*

This command is a synonym for `monotone cert` *`id`* `tag` *`tagname`*.

`monotone testresult` *`id`* `0`
`monotone testresult` *`id`* `1`

These commands are synonyms for `monotone cert` *`id`* `testresult 0` or `monotone cert` *`id`* `testresult 1`.

5.7 Packet I/O

Monotone can produce and consume data in a convenient, portable form called *packets*. A packet is a sequence of ASCII text, wrapped at 70-columns and easily sent through email or other transports. If you wish to manually transmit a piece of information – for example a public key – from one monotone database to another, it is often convenient to read and write packets.

Note: earlier versions of monotone queued and replayed packet streams for their networking system. This older networking system is deprecated and will be removed in a future version, as the netsync protocol has several properties which make it advantageous as a communication system. However, the packet i/o facility will remain in monotone as a utility for moving individual data items around manually.

`monotone certs` *`id`*

This command prints out an `rcert` packet for each cert in your database associated with *id*. These can be used to transport certificates safely between monotone databases.

`monotone fdata` *`id`*
`monotone mdata` *`id`*
`monotone rdata` *`id`*

These commands print out an `fdata`, `mdata` or `rdata` packet for the file, manifest or revision *id* in your database. These can be used to transport files, manifests or revisions, in their entirety, safely between monotone databases.

`monotone fdelta` *`id1 id2`*
`monotone mdelta` *`id1 id2`*

These commands print out an `fdelta` or `mdelta` packet for the differences between file or manifest versions *id1* and *id2*, in your database. These can be used to transport file or manifest differences safely between monotone databases.

`monotone privkey` *`keyid`*
`monotone pubkey` *`keyid`*

These commands print out an `privkey` or `pubkey` packet for the RSA key *keyid*. These can be used to transport public or private keys safely between monotone databases.

`monotone read`
`monotone read` *`file1 file2...`*

This command reads packets from files or `stdin` and stores them in your database.

5.8 Database

`monotone set` *`domain name value`*

Associates the value *value* to *name* in domain *domain*. See Section 3.6 [Vars], page 52 for more information.

`monotone unset` *`domain name`*

Deletes any value associated with *name* in *domain*. See Section 3.6 [Vars], page 52 for more information.

`monotone db init --db=`*`dbfile`*

This command initializes a new monotone database at '`dbfile`'.

`monotone db rebuild --db=`*`dbfile`*

This command rebuilds the ancestry graph of the monotone database at '`dbfile`', which may become necessary if future bugs in monotone allow invalid changesets to be saved in your database. This command is destructive, and you should make a backup copy of your database before running it. It will preserve the contents of each revision, but it will lose rename history. Use it carefully, and only after understanding Section 7.3 [Rebuilding ancestry], page 123. Note that it will make your history incompatible with that of anyone else working on the same project! Read Section 7.3 [Rebuilding ancestry], page 123.

`monotone db info --db=`*`dbfile`*

This command prints information about the monotone database '`dbfile`', including its schema version and various table size statistics.

`monotone db version --db=`*`dbfile`*

This command prints out just the schema version of the monotone database '`dbfile`'.

`monotone db dump --db=`*`dbfile`*

This command dumps an SQL statement representing the entire state of '`dbfile`' to the standard output stream. It is a very low-level command, and produces the most "recoverable" dumps of your database possible. It is sometimes also useful when migrating databases between variants of the underlying sqlite database format.

`monotone db load --db=`*`dbfile`*

This command applies a raw SQL statement, read from the standard input stream, to the database '`dbfile`'. It is most useful when loading a database dumped with the `dump` command.

Note that when reloading a dumped database, the schema of the dumped database is *included* in the dump, so you should not try to `init` your database before a `load`.

`monotone db migrate --db=`*`dbfile`*

This command attempts to migrate the database '`dbfile`' to the newest schema known by the version of monotone you are currently running. If the migration fails, no changes should be made to the database.

If you have important information in your database, you should back up a copy of it before migrating, in case there is an untrapped error during migration.

`monotone db check --db=`*`dbfile`*

Monotone always works hard to verify the data it creates and accesses. For instance, if you have hard drive problems that corrupt data in monotone's database, and you attempt to retrieve this data, then monotone will notice the problem and stop, instead of silently giving you garbage data.

However, it's also nice to notice such problems early, and in rarely used parts of history, while you still have backups. That's what this command is for. It systematically checks the database '`dbfile`' to ensure that it is complete and consistent. The following problems are detected:

- missing files that are referenced by their SHA1 hash from some manifest but do not exist in the database. This is a serious problem; it means that your history is not fully reconstructible. You can fix it by finding the file with the given hash, and loading it into your database with `fload`.
- unreferenced files that exist in the database but are not referenced by their SHA1 hash from any existing manifest. In itself, this only indicates some wasted space, and is not a problem; it's possible it could arise under normal use (for instance, in some strange corner cases following an incomplete netsync). It could also arise, though, as a symptom of some other more serious problem.
- missing manifests that are referenced by their SHA1 hash from some revision but do not exist in the database. This is a serious problem; it means that your history is not fully reconstructible. You can fix it by finding a database containing the manifest, and using `mdata` on that database to create a manifest data packet, which can be loaded into your database with `read`.
- unreferenced manifests that exist in the database but are not referenced by their SHA1 hash from any existing revision. In itself, this only indicates some wasted space, and is not a problem; it's possible it could arise under normal use (for instance, if you have run `db kill_rev_locally`, or in some strange-but-harmless corner cases following an incomplete netsync). It could also arise, though, as a symptom of some other more serious problem.
- incomplete manifests that exist in the database but contain references to files that do not exist in the database. For diagnosis and solution, see "missing files" above.
- missing revisions that are referenced by their SHA1 hash from some other revision or revision cert but do not exist in the database. This may be a serious problem; it may indicate that your history is not fully reconstructible (if the reference is from another revision) or that someone is creating bogus certs (if the reference is from a cert). You can fix it by finding a database containing the revision, and using `rdata` on that database to create a revision data packet, which can be loaded into your database with `read`.
- incomplete revisions that exist in the database but contain references to

missing manifests, incomplete manifests or missing revisions. This always occurs with some more detailed error; see above.

- revisions with mismatched parents that disagree with the cached revision ancestry on their parent revisions. This may cause problems in using the database, and suggests the presence of a bug in monotone's caching system, but does not involve data loss.
- revisions with mismatched children that disagree with the cached revision ancestry on their child revisions. This may cause problems in using the database, and suggests the presence of a bug in monotone's caching system, but does not involve data loss.
- revisions with bad history that exist in the database but fail monotone's normal sanity checks for consistent and correct history. This is a serious problem; it indicates that your history record is somehow malformed. This should not be possible, since monotone carefully checks every revision before storing it into the database, but if it does, then please request assistance on the monotone mailing list. Fixing this generally means you may lose some history — for instance, renames may be degraded into delete/add pairs — but the actual contents of every revision will still be reproducible.
- revisions with missing certs that exist in the database lacking at least one author, branch, changelog or date cert. All revisions are expected to have at least one of each of these certs. In itself, this is not necessarily a problem, but it is peculiar, and some operations such as netsync may behave strangely.
- revisions with mismatched certs that exist in the database with differing numbers of author, changelog and date certs. These certs are expected to appear together, as each revision committed should have an author, changelog and date associated with it. In itself, this is not a problem, but it is peculiar. All operations should behave normally.
- missing keys that have been used to sign certs but do not exist in the database. In itself, this is not a problem, except that monotone will ignore any certs signed by the missing key. You can fix it by finding a database containing the key in question, and using `pubkey` on that database to create a public key packet, which can be loaded into your database with `read`.
- certs with bad signatures that exist in the database with signatures that are invalid. In itself, this is not a problem, except that monotone will ignore any such certs. You may also wish to find out who is creating certs with bad signatures; it may indicate some kind of security attack.
- certs with unchecked signatures that exist in the database but cannot have their signatures checked because the signing key is missing. In itself, this is not a problem, except that monotone will ignore any certs signed by the missing key. You can fix it by finding a database containing the key in question, and using `pubkey` on that database to create a public key packet, which can be loaded into your database with `read`.

This command also verifies that the SHA1 hash of every file, manifest, and revision is correct.

`monotone db kill_rev_locally` *`id`*

This command "kills", i.e., deletes, a given revision, as well as any certs attached to it. It has an ugly name because it is a dangerous command; it permanently and irrevocably deletes historical information from your database. There are a number of caveats:

- It can only be applied to revisions that have no descendents. If you want to kill a revision that has descendents, you must kill all of the descendents first.
- It only removes the revision from your local database (hence the "locally" in the command name). If you have already pushed this revision out to another database, then the next time you pull from that database it may come back again. There is no way to delete a revision from somebody else's database except to ask them to delete it for you.
- It does not actually delete the revision's files or manifest from your database. If you run this command, and then run `db check`, it will note that you have an "unreferenced manifest". If you wish to eliminate this data for good (and thus free up the space), you may use netsync to `pull` from your current database into a new database; this creates a copy of your old database, without the unreferenced data. However, having this data in your database will not cause any problems, and acts as a safety net; if you later realize that you do, after all, need to retrieve the data in *id*, then `db check` will let you see which manifest it was, and with some work you can extract *id*'s data.

`monotone db kill_branch_certs_locally` *`branch`*

This command "kills" a branch by deleting all branch certs with that branch name. You should consider carefully whether you want to use it, because it can irrevocably delete important information. It does not modify or delete any revisions or any of the other certificates on revisions in the branch; it simply removes the branch certificates matching the given branch name. Because of this, it can leave revisions without any branch certificate at all. As with `db kill_rev_locally`, it only deletes the information from your local database; if there are other databases that you sync with which have revisions in this branch, the branch certificates will reappear when you sync, unless the owners of those databases also delete those certificates locally.

`monotone db kill_tag_locally` *`tag`*

This command "kills" a tag by deleting all tag certs with that tag name. You should consider carefully whether you want to use it, because it can irrevocably delete important information. It does not modify or delete any revisions, or any of the other certificates on tagged revisions; it simply removes all tag certificates with the given name. As with `db kill_rev_locally`, it only deletes the information from your local database; if there are other databases that you sync with which have this tag, the tag certificates will reappear when you sync, unless the owners of those databases also delete those certificates locally.

`monotone db execute` *`sql-statement`*

This is a debugging command which executes *sql-statement* against your database, and prints any results of the expression in a tabular form. It can be used to investigate the state of your database, or help diagnose failures.

5.9 Automation

This section contains subcommands of the `monotone automate` command, used for scripting monotone. All give output on stdout; they may also give useful chatter on stderr, including warnings and error messages.

`monotone automate interface_version`

Arguments:
None.

Added in:
0.0

Purpose:
Prints version of the automation interface. Major number increments whenever a backwards incompatible change is made to the `automate` command; minor number increments whenever any change is made (but is reset when major number increments).

Sample output:
```
1.2
```

Output format:
A decimal number, followed by "." (full stop/period), followed by a decimal number, followed by a newline, followed by end-of-file. The first decimal number is the major version, the second is the minor version.

Error conditions:
None.

`monotone automate heads [`*`branch`*`]`

Arguments:
One branch name, *branch*. If none is given, the current default branch is used.

Added in:
0.0

Purpose:
Prints the heads of branch *branch*.

Sample output:
```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:
Zero or more lines, each giving the ID of one head of the given branch. Each line consists of a revision ID, in hexadecimal, followed by a newline. The lines are printed in alphabetically sorted order.

Error conditions:
If the given branch contains no members or does not exist, then no lines are printed.

`monotone automate ancestors` *`rev1`* `[`*`rev2`* `[...]]`

Arguments:

One or more revision IDs, *rev1*, *rev2*, etc.

Added in:

0.2

Purpose:

Prints the ancestors of one or more revisions.

Sample output:

```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:

Zero or more lines, each giving the ID of one ancestor of the given revisions. Each line consists of a revision ID, in hexadecimal, followed by a newline. The lines are printed in alphabetically sorted order.

The output does not include *rev1*, *rev2*, etc., except if *rev2* is itself an ancestor of *rev1*, then *rev2* will be included in the output.

Error conditions:

If any of the revisions do not exist, prints nothing to stdout, prints an error message to stderr, and exits with status 1.

`monotone automate parents` *`rev`*

Arguments:

One revision ID, *rev*.

Added in:

0.2

Purpose:

Prints the immediate parents of a revision. This is like a non-recursive version of `automate ancestors`.

Sample output:

```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:

Zero or more lines, each giving the ID of one parent of the given revision. Each line consists of a revision ID, in hexadecimal, followed by a newline. The lines are printed in alphabetically sorted order.

Error conditions:

If the given revision *rev* does not exist, prints nothing to stdout, prints an error message to stderr, and exits with status 1.

`monotone automate descendents` *`rev1`* `[`*`rev2`* `[...]]`

Arguments:

One or more revision IDs, *rev1*, *rev2*, etc.

Added in:

0.1

Purpose:

Prints the descendents of one or more revisions.

Sample output:

```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:

Zero or more lines, each giving the ID of one descendent of the given revisions. Each line consists of a revision ID, in hexadecimal, followed by a newline. The lines are printed in alphabetically sorted order.

The output does not include *rev1*, *rev2*, etc., except that if *rev2* is itself a descendent of *rev1*, then *rev2* will be included in the output.

Error conditions:

If any of the revisions do not exist, prints nothing to stdout, prints an error message to stderr, and exits with status 1.

`monotone automate children` *`rev`*

Arguments:

One revision ID, *rev*.

Added in:

0.2

Purpose:

Prints the immediate children of a revision. This is like a non-recursive version of `automate descendents`.

Sample output:

```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:

Zero or more lines, each giving the ID of one child of the given revision. Each line consists of a revision ID, in hexadecimal, followed by a newline. The lines are printed in alphabetically sorted order.

Error conditions:

If the given revision *rev* does not exist, prints nothing to stdout, prints an error message to stderr, and exits with status 1.

`monotone automate graph`

Arguments:

None.

Added in:

0.2

Purpose:

Prints out the complete ancestry graph of this database.

Sample output:

```
0c05e8ec9c6af4224672c7cc4c9ef05ae8bdb794
27ebcae50e1814e35274cb89b5031a423c29f95a 5830984dec5c41d994bcadfeab4bf1
4e284617c80bec7da03925062a84f715c1b042bd 27ebcae50e1814e35274cb89b5031a
```

Output format:

Zero or more lines, each giving ancestry information for one revision. Each line begins with a revision ID. Following this are zero or more space-prefixed revision IDs. Each revision ID after the first is a parent (in the sense of `automate parents`) of the first. For instance, in the above sample output, 0c05e8ec9c6af4224672c7cc4c9ef05ae8bdb794 is a root node, 27ebcae50e1814e35274cb89b5031a423c29f95a has one parent, and 4e284617c80bec7da03925062a84f715c1b042bd has two parents, i.e., is a merge node.

The output as a whole is alphabetically sorted by line; additionally, the parents within each line are alphabetically sorted.

Error conditions:

None.

`monotone automate erase_ancestors [`*rev1*` [`*rev2*` [...]]]`

Arguments:

One or more revision IDs, *rev1*, *rev2*, etc.

Added in:

0.1

Purpose:

Prints all arguments, except those that are an ancestor of some other argument. One way to think about this is that it prints the minimal elements of the given set, under the ordering imposed by the "child of" relation. Another way to think of it is if the arguments formed a branch, then we would print the heads of that branch. If there are no arguments, prints nothing.

Sample output:

```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:

Zero or more lines, each giving the ID of one descendent of the given revisions. Each line consists of a revision ID, in hexadecimal, followed by a newline. The lines are printed in alphabetically sorted order.

Error conditions:

If any of the revisions do not exist, prints nothing to stdout, prints an error message to stderr, and exits with status 1.

`monotone automate toposort [`*`rev1`* `[`*`rev2`* `[...]]]`

Arguments:

One or more revision IDs, *rev1*, *rev2*, etc.

Added in:

0.1

Purpose:

Prints all arguments, topologically sorted. I.e., if *rev1* is an ancestor of *rev2*, then *rev1* will appear before *rev2* in the output; if *rev2* is an ancestor of *rev1*, then *rev2* will appear before *rev1* in the output; and if neither is an ancestor of the other, then they may appear in either order. If there are no arguments, prints nothing.

Sample output:

```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:

A list of revision IDs, in hexadecimal, each followed by a newline. Revisions are printed in topologically sorted order.

Error conditions:

If any of the revisions do not exist, prints nothing to stdout, prints an error message to stderr, and exits with status 1.

`monotone automate ancestry_difference` *`new`* `[`*`old1`* `[`*`old2`* `[...]]]`

Arguments:

A "new" revision ID *new*, followed by zero or more "old" revision IDs *old1*, *old2*, etc.

Added in:

0.1

Purpose:

Prints all ancestors of the revision *new*, that are not also ancestors of one of the old revisions. For purposes of this command, "ancestor" is an inclusive term; for example, if *new* is an ancestor of *old1*, it will not be printed; but if *new* is not an ancestor of any of the "old" revisions, then it will be. Similarly, *old1* will never be printed, because it is considered to be an ancestor of itself. The reason for the names is that if *new* a new revision, and *old1*, *old2*, etc. are revisions that you have processed before, then this command tells you which revisions are new since then.

Sample output:

```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:

A list of revision IDs, in hexadecimal, each followed by a newline. Revisions are printed in topologically sorted order.

Error conditions:
If any of the revisions do not exist, prints nothing to stdout, prints an error message to stderr, and exits with status 1.

`monotone automate leaves`

Arguments:
None.

Added in:
0.1

Purpose:
Prints the leaves of the revision graph, i.e. all revision that have no children. This is similar, but not identical to the functionality of `heads`, which prints every revision in a branch, that has no descendents in that branch. If every revision in the database was in the same branch, then they would be identical. Generally, every leaf is the head of some branch, but not every branch head is a leaf.

Sample output:
```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:
Zero or more lines, each a leaf of the revision graph. Each line consists of a revision ID, in hexadecimal, followed by a newline. The lines are printed in alphabetically sorted order.

Error conditions:
None.

`monotone automate select` *`selector`*

Arguments:
One selector (or combined selector).

Added in:
0.2

Purpose:
Print all revisions that match the given selector.

Sample output:
```
28ce076c69eadb9b1ca7bdf9d40ce95fe2f29b61
75156724e0e2e3245838f356ec373c50fa469f1f
```

Output format:
Zero or more lines, each giving the ID of one revision that matches the given selector. Each line consists of a revision ID, in hexadecimal, followed by a newline.

Error conditions:
None.

`monotone automate inventory`

Arguments:

None.

Added in:

1.0

Purpose:

Prints the inventory of every file found in the working copy or its associated base manifest. Each unique path is listed on a line prefixed by three status characters and two numeric values used for identifying renames.

Sample output:

```
  M 0 0 missing
 AP 0 0 added patched
D   0 0 dropped
R   1 0 renamed-from-this
 R  0 1 renamed-to-this
  P 0 0 patched
    0 0 unchanged
  U 0 0 unknown
  I 0 0 ignored

# swapped but not moved

RRP 1 2 unchanged
RRP 2 1 original

# swapped and moved

RR  1 2 unchanged
RR  2 1 original

# rename foo bar; add foo

RAP 1 0 foo
 R  0 1 bar

# rotated but not moved dropped -> missing -> original -> dropped

RRP 1 3 dropped
RRP 2 1 missing
RRP 3 2 original

# rotated and moved

RR  1 3 dropped
RR  2 1 missing
RR  3 2 original

# dropped but not removed and thus unknown

D U 0 0 dropped

# added but removed and thus missing

 AM 0 0 added

# renamed but not moved and thus unknown source and missing target

R U 1 0 original
 RM 0 1 renamed

# moved but not renamed and thus missing source and unknown target

  M 0 0 original
  U 0 0 renamed

# renamed and patched

R   1 0 original
 RP 0 1 renamed
```

Output format:

Each path is printed on its own line, prefixed by three status characters described below. The status is followed by a single space and two numbers, each separated by a single space, used for identifying renames. The numbers are followed by a single space and then the pathname, which includes the rest of the line. Directory paths are identified as ending with the "/" character, file paths do not end in this character.

The three status characters are as follows.

```
column 1 pre-state
      ' ' the path was unchanged in the pre-state
      'D' the path was deleted from the pre-state
      'R' the path was renamed from the pre-state name
column 2 post-state
      ' ' the path was unchanged in the post-state
      'R' the path was renamed to the post-state name
      'A' the path was added to the post-state
column 3 file-state
      ' ' the file is known and unchanged from the current manifest ver
      'P' the file is patched to a new version
      'U' the file is unknown and not included in the current manifest
      'I' the file is ignored and not included in the current manifest
      'M' the file is missing but is included in the current manifest
```

Note that there are 45 possible status code combinations, some of which are not valid, detailed below.

```
'   ' unchanged
'  P' patched (contents changed)
'  U' unknown (exists on the filesystem but not tracked)
'  I' ignored (exists on the filesystem but excluded by lua hook)
'  M' missing (exists in the manifest but not on the filesystem)

' A ' added (invalid, add should have associated patch)
' AP' added and patched
' AU' added but unknown (invalid)
' AI' added but ignored (seems invalid, but may be possible?)
' AM' added but missing from the filesystem

' R ' rename target
' RP' rename target and patched
' RU' rename target but unknown (invalid)
' RI' rename target but ignored (seems invalid, but may be possible?)
' RM' rename target but missing from the filesystem

'D  ' dropped
'D P' dropped and patched (invalid)
'D U' dropped and unknown (still exists on the filesystem)
'D I' dropped and ignored (seems invalid, but may be possible?)
'D M' dropped and missing (invalid)

'DA ' dropped and added (invalid, add should have associated patch)
'DAP' dropped and added and patched
'DAU' dropped and added but unknown (invalid)
'DAI' dropped and added but ignored (seems invalid, but may be possible
'DAM' dropped and added but missing from the filesystem

'DR ' dropped and rename target
'DRP' dropped and rename target and patched
'DRU' dropped and rename target but unknown (invalid)
'DRI' dropped and rename target but ignored (invalid)
'DRM' dropped and rename target but missing from the filesystem

'R  ' rename source
'R P' rename source and patched (invalid)
'R U' rename source and unknown (still exists on the filesystem)
'R I' rename source and ignored (seems invalid, but may be possible?)
'R M' rename source and missing (invalid)
```

```
'RA ' rename source and added (invalid, add should have associated patc
'RAP' rename source and added and patched
'RAU' rename source and added but unknown (invalid)
'RAI' rename source and added but ignored (seems invalid, but may be po
'RAM' rename source and added but missing from the filesystem

'RR ' rename source and target
'RRP' rename source and target and target patched
'RRU' rename source and target and target unknown (invalid)
'RRI' rename source and target and target ignored (seems invalid, but m
'RRM' rename source and target and target missing
```

Full support for versioned directories is not yet complete and the inventory will only list entries for renamed or dropped directories.

Error conditions:

When executed from outside of a working copy directory, prints an error message to stderr, and exits with status 1.

`monotone automate certs` *`id`*

Arguments:

A revision ID *id*, for which any certificates will be printed.

Added in:

1.0

Purpose:

Prints all certificates associated with the given revision ID. Each certificate is contained in a basic IO stanza. For each certificate, the following values are provided:

```
'key'
      a string indicating the key used to sign this certificate.
'signature'
      a string indicating the status of the signature. Possible
      values of this string are:
            'ok'        : the signature is correct
            'bad'       : the signature is invalid
            'unknown'   : signature was made with an unknown key
'name'
      the name of this certificate
'value'
      the value of this certificate
'trust'
      is this certificate trusted by the defined trust metric?
      Possible values of this string are:
            'trusted'   : this certificate is trusted
            'untrusted' : this certificate is not trusted
```

Sample output:

```
      key "emile@alumni.reed.edu"
signature "ok"
     name "author"
    value "emile@alumni.reed.edu"
    trust "trusted"

      key "emile@alumni.reed.edu"
signature "ok"
     name "branch"
    value "net.venge.monotone"
    trust "trusted"

      key "emile@alumni.reed.edu"
signature "ok"
     name "changelog"
    value "propagate from branch 'net.venge.monotone.annotate' (head 76
             to branch 'net.venge.monotone' (head 2490479a4e4e99243fead6
"
    trust "trusted"

      key "emile@alumni.reed.edu"
signature "ok"
     name "date"
    value "2005-05-20T20:19:25"
    trust "trusted"
```

Output format:

All stanzas are formatted by basic_io. Stanzas are seperated by a blank line. Values will be escaped, '\' to '\\' and '"' to '\"'.

Error conditions:

If a certificate is signed with an unknown public key, a warning message is printed to stderr. If the revision specified is unknown or invalid prints an error message to stderr and exits with status 1.

`monotone automate stdio`

Arguments:

none

Added in:

1.0

Purpose:

Allow multiple automate commands to be run from one instance of monotone.

Sample input:

```
l6:leavese
l7:parents40:0e3171212f34839c2e3263e7282cdeea22fc5378e
```

Input format:

The input is a series of commands of the form 'l' <token> [<token> ...] 'e', where <token> = <size> colon <string> . Characters between the ending 'e' of one commad and the beginning 'l' of the next are ignored, but characters other than '\n' should not be present as this space is reserved.

Sample output:

```
0:0:l:205:0e3171212f34839c2e3263e7282cdeea22fc5378
1f4ef73c3e056883c6a5ff66728dd764557db5e6
2133c52680aa2492b18ed902bdef7e083464c0b8
23501f8afd1f9ee037019765309b0f8428567f8a
2c295fcf5fe20301557b9b3a5b4d437b5ab8ec8c
1:0:l:41:7706a422ccad41621c958affa999b1a1dd644e79
```

Output format:

<command number>:<err code>:<last?>:<size>:<output>

<command number> is a decimal number specifying which command this output is from. It is 0 for the first command, and increases by one each time.

<err code> is 0 for success, 1 for a syntax error, and 2 for any other error.

<last?> is 'l' if this is the last piece of output for this command, and 'm' if there is more output to come.

<size> is the number of bytes in the output.

<output> is the output of the command. This will never exceed 1024 bytes. If a command produces more than 1024 bytes of output, it will be split into multiple peices, with all but the last having the <last?> field set to 'm'.

Error conditions:

If a badly formatted command is recieved, prints an error message to standard error and exits with nonzero status. Errors in the commands run through this interface do not affect the exit status. Instead, the <err code> field in the output is set to 2, and the output of the command becomes whatever error message would have been given. The output of an invalid command line will have an error code of 1, and the output text will be a help message.

`monotone automate get_revision`
`monotone automate get_revision` *`id`*

Arguments:

Specifying the option *id* argument outputs the changeset information for the specified *id*. Otherwise, *id* is determined from the working directory.

Added in:

1.0

Purpose:

Prints changeset information for the specified revision id.

There are several changes that are described; each of these is described by a different basic_io stanza. The first string pair of each stanza indicates the type of change represented.

Possible values of this first value are along with an ordered list of basic_io formatted string pairs that will be provided are:

```
'old_revision'
      represents a parent revision.
      format: ('old_revision', revision id)
'new_manifest'
      represents the new manifest associated with the revision.
      format: ('new_manifest', manifest id)
'old_manifest'
      represents a manifest associated with a parent revision.
      format: ('old_manifest', manifest id)
'patch'
      represents a file that was modified.
      format: ('patch', filename), ('from', file id), ('to', file id)
'add_file'
      represents a file that was added.
      format: ('add_file', filename)
'delete_file'
      represents a file that was deleted.
      format: ('delete_file', filename)
'delete_dir'
      represents a directory that was deleted.
      format: ('delete_dir', filename)
'rename_file'
      represents a file that was renamed.
      format: ('rename_file', old filename), ('to', new filename)
'rename_dir'
      represents a directory that was renamed.
      format: ('rename_dir', old filename), ('to', new filename)
```

Sample output:

```
new_manifest [bfe2df785c07bebeb369e537116ab9bb7a4b5e19]

old_revision [429fea55e9e819a046843f618d90674486695745]
old_manifest [b71855116f7049ac663102c0cb628653ffe316d1]

patch "ChangeLog"
 from [7dc21d3a46c6ecd94685ab21e67b131b32002f12]
   to [234513e3838d423b24d5d6c98f70ce995c8bab6e]

patch "std_hooks.lua"
 from [0408707bb6b97eae7f8da61af7b35364dbd5a189]
```

```
          to [d7bd0756c48ace573926197709e53eb24dae5f5f]
```

Output format:

All stanzas are formatted by basic_io. Stanzas are seperated by a blank line. Values will be escaped, '\' to '\\' and '"' to '\"'.

Error conditions:

If the revision specified is unknown or invalid prints an error message to stderr and exits with status 1.

`monotone automate get_manifest`
`monotone automate get_manifest` *`id`*

Arguments:

Specifying the optional *id* argument outputs the maifest for the specified *id*. Otherwise, *id* is determined from the working directory.

Added in:

1.0

Purpose:

Prints the contents of the manifest associated with the given manifest ID.

Sample output:

```
e3915658cb464d05f21332e03d30dca5d94fe776  .htaccess
00ff89c69e7a8f6a0f48c6503168d9b62a0cfeb0  .mt-attrs
80d8f3f75c9b517ec462233e155f7dfb93379f67  AUTHORS
fc74a48c7f73eedcbe1ea709755fbe819b29736c  ChangeLog
dfac199a7539a404407098a2541b9482279f690d  LICENSE
440eec971e7bb61ccbb61634deb2729bb25931cd  README
e0ea26c666b37c5f98ccf80cb933d021ee55c593  TODO
b28ece354969314ce996f3030569215d685973d6  branch.psp
1fdb62e05fb2a9338d2c72ddc58de3ab2b3976fe  common.py
64cb5898e3a026b4782c343ca4386585e0c3c275  config.py.example
7152c3ff110418aca5d23c374ea9fb92a0e98379  error.psp
5d8536100fdf51d505b6f20bc9c16aa78d4e86a8  fileinbranch.psp
981df124a0b5655a9f78c42504cfa8c6f02b267a  headofbranch.psp
a43d0588a69e622b2afc681678c2a5c3b3b1f342  help.psp
18a8bffc8729d7bfd71d2e0cb35a1aed1854fa74  html.py
c621827db187839e1a7c6e51d5f1a7f6e0aa560c  index.psp
708b61436dce59f47bd07397ce96a1cfabe81970  monotone.py
a02b1c161006840ea8685e461fd07f0e9bb145a3  revision.psp
027515fd4558abf317d54c437b83ec6bc76e3dd8  rss_feed.gif
638140d6823eee5844de37d985773be75707fa25  tags.psp
be83f459a152ffd49d89d69555f870291bc85311  tarofbranch.psp
fb51955563d64e628e0e67e4acca1a1abc4cd989  utility.py
8d04b3fc352a860b0e3240dcb539c1193705398f  viewmtn.css
7cb5c6b1b1710bf2c0fa41e9631ae43b03424a35  viewmtn.py
530290467a99ca65f87b74f653bf462b28c6cda9  wrapper.py
```

Output format:

The output format is one line for each file in the manifest. Each line begins with a 40 character file ID, followed by two space characters (' ') and then the filename.

Error conditions:

If the manifest ID specified is unknown or invalid prints an error message to stderr and exits with status 1.

`monotone automate get_file` *`id`*

Arguments:

The *id* argument specifies the file to be output.

Added in:

1.0

Purpose:

Prints the contents of the specified file.

Sample output:

```
If you've downloaded a release, see INSTALL for installation
instructions.  If you've checked this out, the generated files are not
included, and you must use "autoreconf --install" to create them.

"make html" for docs, or read the .info file and / or man page.
```

Output format:

The file contents are output without modification.

Error conditions:

If the file id specified is unknown or invalid prints an error message to stderr and exits with status 1.

5.10 RCS

`monotone rcs_import` *`filename...`*

This command imports all the file versions in each RCS file listed in *filename...*. These files should be raw RCS files, ending in `,v`. Monotone parses them directly and inserts them into your database. Note that this does not do any revision reconstruction, and is only useful for debugging.

`monotone cvs_import` *`pathname`*

This command imports all the file versions in each RCS file found in the tree of files starting at *pathname*, then reconstructs the tree-wide history of logical changes by comparing RCS time stamps and change log entries. For each logical tree-wide change, monotone synthesizes a manifest and revision, and commits them (along with all associated file deltas) to your database. It also copies all change log entries, author identifiers, and date stamps to manifest certificates.

In normal use, *pathname* will be a CVS module, though it is possible to point it at a directory within a module as well. Whatever directory you point it at will become the root of monotone's version of the tree.

6 Hook Reference

Monotone's behavior can be customized and extended by writing *hook functions*, which are written in the Lua programming language. At certain points in time, when monotone is running, it will call a hook function to help it make a decision or perform some action. If you provide a hook function definition which suits your preferences, monotone will execute it. This way you can modify how monotone behaves.

You can put new definitions for any of these hook functions in a file '`$HOME/.monotone/monotonerc`', or in your working copy in '`MT/monotonerc`', both of which will be read every time monotone runs. Definitions in '`MT/monotonerc`' shadow (override) definitions made in your '`$HOME/.monotone/monotonerc`'. You can also tell monotone to interpret extra hook functions from any other *file* using the '`--rcfile=`*`file`*' option; hooks defined in files specified on the command-line will shadow hooks from the the automatic files. By specifying '`--rcfile=`*`directory`*' you can automatically load all the files contained into *directory*.

Monotone also makes available to hook writers a number of helper functions exposing functionality not available with standard lua.

For the complete source of the default hooks see Appendix A [Default hooks], page 131.

6.1 Hooks

This section documents the existing hook functions and their default definitions.

`note_commit (`*`new_id, revision, certs`*`)`

Called by monotone after the version *new_id* is committed. The second parameter, *revision* is the text of the revision, what would be given by "monotone cat revision *new_id*". The third parameter, *certs*, is a lua table containing the set of certificate names and values committed along with this version. There is no default definition for this hook.

Note that since the *certs* table does not contain cryptographic or trust information, and only contains one entry per cert name, it is an incomplete source of information about the committed version. This hook is only intended as an aid for integrating monotone with informal commit-notification systems such as mailing lists or news services. It should not perform any security-critical operations.

`note_netsync_revision_received (`*`new_id, revision, certs`*`)`

Called by monotone after the revision *new_id* is recieved through netsync. *revision* is the text of the revision, what would be given by "monotone cat revision *new_id*". *certs* is a lua table containing one subtable for each cert attached to the revision *new_id*. These subtables have fields named "key", "name", and "value", containing the signing key for the cert, the name of the cert, and the value of the cert. There is no default definition for this hook.

`note_netsync_cert_received (`*`rev_id, key,`*
name, value)

Called by monotone after a cert is received through netsync, iff the revision that the cert is attached to was not also received in the same netsync operation. *rev_id* is the revision id that the cert is attached to, *key* is the key that the cert is signed with, *name* is the name of the cert, and *value* is the cert value. There is no default definition for this hook.

`note_netsync_pubkey_received (`*`keyname`*`)`

Called by monotone after a pubkey is received through netsync. *keyname* is the name of the key received. There is no default definition for this hook.

`get_branch_key (`*`branchname`*`)`

Returns a string which is the name of an RSA private key used to sign certificates in a particular branch *branchname*. There is no default definition for this hook. The command-line option '`--key=`*`keyname`*' overrides any value returned from this hook function. If you have only one private key in your database, you do not need to define this function or provide a '`--key=`*`keyname`*' option; monotone will guess that you want to use the unique private key.

`get_passphrase (`*`keypair_id`*`)`

Returns a string which is the passphrase used to encrypt the private half of *keypair_id* in your database, using the ARC4 symmetric cipher. *keypair_id* is a Lua string containing the label that you used when you created your key — something like `"nicole@example.com"`. This hook has no default definition.

If this hook is not defined or returns false, monotone will prompt you for a passphrase each time it needs to use a private key.

`get_author (`*`branchname`*`)`

Returns a string which is used as a value for automatically generated `author` certificates when you commit changes to *branchname*. Generally this hook remains undefined, and monotone selects your signing key name for the author certificate. You can use this hook to override that choice, if you like.

This hook has no default definition, but a possible definition might be:

```
function get_author(branchname)
        local user = os.getenv("USER")
        local host = os.getenv("HOSTNAME")
        if ((user == nil) or (host == nil)) then return nil end
        return string.format("%s@%s", user, host)
end
```

`edit_comment (`*`commentary`*`, `*`user_log_message`*`)`

Returns a log entry for a given set of changes, described in *commentary*. The commentary is identical to the output of `monotone status`. This hook is intended to interface with some sort of editor, so that you can interactively document each change you make. The result is used as the value for a `changelog` certificate, automatically generated when you commit changes.

The contents of '`MT/log`' are read and passed as *user_log_message*. This allows you to document your changes as you proceed instead of waiting until you are ready to commit. Upon a successful commit, the contents of '`MT/log`' are erased setting the system up for another edit/commit cycle.

For the default definition of this hook, see Appendix A [Default hooks], page 131.

`persist_phrase_ok ()`

Returns `true` if you want monotone to remember the passphrase of a private key for the duration of a single command, or `false` if you want monotone to prompt you for a passphrase for each certificate it generates. Since monotone often generates several certificates in quick succession, unless you are very concerned about security you probably want this hook to return `true`.

The default definition of this hook is:

```
function persist_phrase_ok()
        return true
end
```

`non_blocking_rng_ok ()`

Returns `true` if you are willing to let monotone use the system's non-blocking random number generator, such as '`/dev/urandom`', for generating random values during cryptographic operations. This diminishes the cryptographic strength of such operations, but speeds them up. Returns `false` if you want to force monotone to always use higher quality random numbers, such as those from '`/dev/random`'.

The default definition of this hook is:

```
function non_blocking_rng_ok()
        return true
end
```

`use_inodeprints ()`

Returns `true` if you want monotone to automatically enable Section 3.4 [Inodeprints], page 50 support in all working copies. Only affects working copies created after you modify the hook.

The default definition of this hook is:

```
function use_inodeprints()
        return false
end
```

`get_netsync_read_permitted (`*`branch, identity`*`)`

Returns `true` if a peer authenticated as key *identity* should be allowed to read from your database certs, revisions, manifests, and files associated with *branch*; otherwise `false`. This hook has no default definition, therefore the default behavior is to deny all reads.

If a client connects anonymously, this hook will be called with a *identity* of `nil`.

Note that the *identity* value is a key ID (such as "`graydon@pobox.com`") but will correspond to a *unique* key fingerprint (hash) in your database. Monotone will not permit two keys in your database to have the same ID. Make sure you confirm the key fingerprints of each key in your database, as key ID strings are "convenience names", not security tokens.

`get_netsync_write_permitted (`*`identity`*`)`

Returns `true` if a peer authenticated as key *identity* should be allowed to write into your database certs, revisions, manifests, and files; otherwise `false`. This hook has no default definition, therefore the default behavior is to deny all writes.

If a client connects anonymously, it will be unconditionally denied write access; this hook will *not* be called with a *identity* of `nil`.

Note that the *identity* value is a key ID (such as "`graydon@pobox.com`") but will correspond to a *unique* key fingerprint (hash) in your database. Monotone will not permit two keys in your database to have the same ID. Make sure you confirm the key fingerprints of each key in your database, as key ID strings are "convenience names", not security tokens.

`ignore_file (`*`filename`*`)`

Returns `true` if *filename* should be ignored while adding, dropping, or moving files. Otherwise returns `false`. This is most important when performing recursive actions on directories, which may affect multiple files simultaneously. For the default definition of this hook, see Appendix A [Default hooks], page 131.

`ignore_branch (`*`branchname`*`)`

Returns `true` if *branchname* should be ignored while listing branches. Otherwise returns `false`. This hook has no default definition, therefore the default behavior is to list all branches.

`get_revision_cert_trust (`*`signers, id, name, val`*`)`

Returns whether or not you *trust* the assertion *name*=*value* on a given revision *id*, given a valid signature from all the keys in *signers*. The *signers* parameter is a table containing all the key names which signed this cert, the other three parameters are strings.

The default definition of this hook simply returns `true`, which corresponds to a form of trust where every key which is defined in your database is trusted. This is a *weak* trust setting; you should change it to something stronger. A possible example of a stronger trust function (along with a utility function for computing the intersection of tables) is the following:

```
function intersection(a,b)
   local s={}
   local t={}
   for k,v in pairs(a) do s[v] = 1 end
   for k,v in pairs(b) do if s[v] ~= nil then table.insert(t,v) end end
   return t
end

function get_revision_cert_trust(signers, id, name, val)
   local trusted_signers = { "bob@happyplace.example.com",
                             "friend@trustedplace.example.com",
                             "myself@home.example.com" }
   local t = intersection(signers, trusted_signers)

   if t == nil then return false end

   if    (name ~= "ancestor" and table.getn(t) >= 1)
      or (name == "ancestor" and table.getn(t) >= 2)
   then
      return true
   else
      return false
   end
end
```

In this example, any revision certificate is trusted if it is signed by at least one of three "trusted" keys, unless it is an `ancestor` certificate, in which case it must be signed by *two* or more trusted keys. This is one way of requiring that ancestry assertions go through an extra "reviewer" before they are accepted.

`accept_testresult_change (`*`old_results`*`, `*`new_results`*`)`

This hook is used by the update algorithm to determine whether a change in test results between update source and update target is acceptable. The hook is called with two tables, each of which maps a signing key – representing a particular testsuite – to a boolean value indicating whether or not the test run was successful. The function should return `true` if you consider an update from the version carrying the *old_results* to the version carrying the *new_results* to be acceptable.

The default definition of this hook follows:

```
function accept_testresult_change(old_results, new_results)
   for test,res in pairs(old_results)
   do
      if res == true and new_results[test] ~= true
      then
 return false
      end
   end
   return true
end
```

This definition accepts only those updates which preserve the set of `true` test results from update source to target. If no rest results exist, this hook has no affect; but once a `true` test result is present, future updates will require it. If you want a more lenient behavior you must redefine this hook.

`merge2 (`*`left, right`*`)`
: Returns a string, which should be the merger of the 2 provided strings, which are the contents of the *left* and *right* nodes of a file fork which monotone was unable to automatically merge. The merge should either call an intelligent merge program or interact with the user. For the default definition of this hook, see Appendix A [Default hooks], page 131.

`get_preferred_merge2_command(`*`tbl`*`)`
: Returns the results of running an external merge on two strings. *tbl* wraps up the various arguments for each merge command and is always provided by [merge2], page 110. If there is a particular editor that you would like to use to perform merge2 operations, override this hook to specify it.

`merge3 (`*`ancestor, left, right`*`)`
: Returns a string, which should be the merger of the 3 provided strings, which are the contents of *left* and *right* nodes, and least common *ancestor*, of a file fork which monotone was unable to automatically merge. This hook delegates the actual merge to the result of [get_preferred_merge3_command], page 110. For the default definition of this hook, see Appendix A [Default hooks], page 131.

`get_preferred_merge3_command(`*`tbl`*`)`
: Returns the results of running an external merge on three strings. *tbl* wraps up the various arguments for each merge command and is always provided by [merge3], page 110. If there is a particular editor that you would like to use to perform merge3 operations, override this hook to specify it.

`expand_selector (`*`str`*`)`
: Attempts to expand *str* as a selector. Expansion generally means providing a type prefix for the selector, such as `a:` for authors or `d:` for dates. Expansion may also mean recognizing and interpreting special words such as `yesterday` or `6 months ago` and converting them into well formed selectors. For more detail on the use of selectors, see Section 3.1 [Selectors], page 44. For the default definition of this hook, see Appendix A [Default hooks], page 131.

`expand_date (`*`str`*`)`
: Attempts to expand *str* as a date expression. Expansion means recognizing and interpreting special words such as `yesterday` or `6 months ago` and converting them into well formed date expressions. For more detail on the use of selectors, see Section 3.1 [Selectors], page 44. For the default definition of this hook, see Appendix A [Default hooks], page 131.

`get_system_linesep ()`
: Returns a string which defines the default system line separator. This should be one of the strings `CR`, `LF`, or `CRLF`. The system line separator may be used when reading or writing data to the terminal, or otherwise interfacing with the

user. The system line separator is not used to convert files in the working copy; use `get_linesep_conv` for converting line endings in the working copy.

This hook has no default definition. For more information on line ending conversion, see the section on Section 7.1 [Internationalization], page 116.

`get_linesep_conv (`*`filename`*`)`

Returns a table which contains two strings. The first string in the return value is the name of a line ending convention to use for the "internal" representation of *filename*. The second string in the return value is the name of a line ending convention to use for the "external" representation of *filename*. Line ending conventions should be one of the strings `CR`, `LF`, or `CRLF`.

When *filename* is read from the working copy, it is run through line ending conversion from the external form to the internal form. When *filename* is written to the working copy, it is run through line ending conversion from the internal form to the external form. SHA1 values are calculated from the internal form of *filename*. It is your responsibility to decide which line ending conversions your work will use.

This hook has no default definition; monotone's default behavior is to keep external and internal forms byte-for-byte identical. For more information on line ending conversion, see the section on Section 7.1 [Internationalization], page 116.

`get_charset_conv (`*`filename`*`)`

Returns a table which contains two strings. The first string in the return value is the name of a character set to use for the "internal" representation of *filename*. The second string in the return value is the name of a character set to use for the "external" representation of *filename*.

When *filename* is read from the working copy, it is run through character set conversion from the external form to the internal form. When *filename* is written to the working copy, it is run through character set conversion from the internal form to the external form. SHA1 values are calculated from the internal form of *filename*. It is your responsibility to decide which character set conversions your work will use.

This hook has no default definition; monotone's default behavior is to keep external and internal forms byte-for-byte identical. For more information on character set conversion, see the section on Section 7.1 [Internationalization], page 116.

`attr_functions [`*`attribute`*`] (`*`filename`*`, `*`value`*`)`

This is not a hook function, but a *table* of hook functions. Each entry in the table `attr_functions`, at table entry *attribute*, is a function taking a file name *filename* and a attribute value *value*. The function should "apply" the attribute to the file, possibly in a platform-specific way.

Persistent attributes are stored in the '`.mt-attrs`', in your working copy and manifest. If such a file exists, hook functions from this table are called for each triple found in the file, after any command which modifies the working copy. This facility can be used to extend monotone's understanding of files with platform-specific attributes, such as permission bits, access control lists, or special file types.

By default, there is only one entry in this table, for the `execute` attribute. Its definition is:

```
attr_functions["execute"] =
  function(filename, value)
        if (value == "true") then
         make_executable(filename)
      end
   end
```

`attr_init_functions [`*`attribute`*`] (`*`filename`*`)`

This is not a hook function, but a *table* of hook functions. Each entry in the table `attr_init_functions`, at table entry *attribute*, is a function taking a file name *filename*. Each function defines the attributes that should be stored in '`.mt-attrs`' for the given *filename*. This table of hook functions is called once for each file during an *add*.

By default, there are only two entries in this table, for the `execute` and `manual_merge` attributes. Their definition is:

```
attr_init_functions["execute"] =
   function(filename)
      if (is_executable(filename)) then
        return "true"
      else
        return nil
        end
  end
attr_init_functions["manual_merge"] =
   function(filename)
      if (binary_file(filename)) then
        return "true" -- binary files must merged manually
      else
        return nil
      end
   end
```

The `binary_file` function is also defined as a lua hook. See Appendix A [Default hooks], page 131.

`external_diff (`*`file_path`*`, `*`old_data`*`, `*`new_data`*`, `*`is_binary`*`,`
diff_args, *old_rev*, *new_rev*)

Called for each file when `diff` is given the '`--external`' command. *file_path* is the pathname of the file that is being diffed. *old_data* and *new_data* are the data contents of the old and the new file. If the data is binary, *is_binary* will be true, otherwise false. *old_rev* and *new_rev* are the revision IDs of the old and new data.

If an extra arguments are given via '`--diff-args`', the string will be passed in as *diff_args*. Otherwise *diff_args* will be nil.

The default implementation of this hook calls the program `diff`, and if '`--diff-args`' were not passed, takes default arguments from the lua variable `external_diff_default_args`. You can override this variable in your configuration file, without overriding the whole hook.

6.2 Additional Lua Functions

This section documents the additional lua functions made available to hook writers.

`existonpath(`*`possible_command`*`)`

This function receives a string containing the name of an external program and returns 0 if it exists on path and is executable, -1 otherwise. As an example, `existonpath("xxdiff")` returns 0 if the program xxdiff is available. On windows, this function automatically appends ".exe" to the program name. In the previous example, `existonpath` would search for "xxdiff.exe".

`guess_binary(`*`filespec`*`)`

Returns true if the file appears to be binary, i.e. contains one or more of the following characters:

```
0x00 thru 0x06
0x0E thru 0x1a
0x1c thru 0x1f
```

`include(`*`scriptfile`*`)`

This function tries to load and execute the script contained into scriptfile. It returns true for success and false if there is an error.

`includedir(`*`scriptpath`*`)`

This function loads and executes in alphabetical order all the scripts contained into the directory scriptpath. If one of the scripts has an error, the functions doesn't process the remaining scripts and immediately returns false.

`is_executable(`*`filespec`*`)`

This function returns true if the file is executable, false otherwise. On windows this function returns always false.

`kill(`*`pid`*` [, `*`signal`*`])`

This function calls the kill() C library function on posix systems and TerminateProcess on Win32 (in that case *pid* is the process handle). If the optional *signal* parameter is missing, SIGTERM will be used. Returns 0 on succes, -1 on error.

`make_executable(`*`filespec`*`)`

This function marks the named file as executable. On windows has no effect.

`mkstemp(`*`template`*`)`

Like its C library counterpart, mkstemp creates a unique name and returns a file descriptor for the newly created file. The value of template should be a pointer to a character buffer loaded with a null-terminated string that consists of contiguous, legal file ad path name characters followed by six Xs. The function mkstemp replaces the Xs by an alpha-numeric sequence that is chosen to ensure that no file in the chosen directory has that name. Furthermore, subsequent calls to mkstemp within the same process each yield different file names. Unlike other implementations, monotone mkstemp allows the template string to contain a complete path, not only a filename, allowing users to create temporary files outside the current directory.

Important notice:

To create a temporary file, you must use the `temp_file()` function, unless you need to run monotone with the '`--nostd`' option. `temp_file()` builds on `mkstemp()` and creates a file in the standard TMP/TEMP directories. For the definition of `temp_file()`, see Appendix A [Default hooks], page 131.

`sleep(`*`seconds`*`)`
: Makes the calling process sleep for the specified number of seconds.

`spawn(`*`executable`*` [, `*`args`*` ...])`
: Starts the named executable with the given arguments. Returns the process pid on Posix systems, the process handle on Win32 or -1 if there was an error. Calls fork/execvp on Posix, CreateProcess on Win32.

  Important notice:
  To spawn a process and wait for its completion, use the `execute()` function, unless you need to run monotone with the '`--nostd`' option. `execute()` builds on `spawn()` and `wait()` in a standardized way.

`wait(`*`pid`*`)`
: Wait until the process with given pid (process handle on Win32) exits. Returns two values: a result value and the exit code of the waited-for process. The exit code is meaningful only if the result value is 0.

7 Special Topics

This chapter describes some "special" issues which are not directly related to monotone's *use*, but which are occasionally of interest to people researching monotone or trying to learn the specifics of how it works. Most users can ignore these sections.

7.1 Internationalization

Monotone initially dealt with only ASCII characters, in file path names, certificate names, key names, and packets. Some conservative extensions are provided to permit internationalized use. These extensions can be summarized as follows:

- Monotone uses GNU gettext to provide localized progress and error messages. Translations may or may not exist for your locale, but the infrastructure is present to add them.
- All command-line arguments are mapped from your local character set to UTF-8 before processing. This means that monotone can *only* handle key names, file names and certificate names which map cleanly into UTF-8.
- Monotone's control files are stored in UTF-8. This includes: revisions and manifests, both inside the database and when written to the '`MT/`' directory of the working copy; the '`MT/options`' and '`MT/work`' files; and the '`.mt-attrs`' file. Converting these files to any other character set will cause monotone to break; do not do so.
- File path names in the working copy are converted to the locale's character set (determined via the LANG or CHARSET environment variables) before monotone interacts with the file system. If you are accustomed to being able to use file names in your locale's character set, this should "just work" with monotone.
- Key and cert names, and similar "name-like" entities are subject to some cleaning and normalization, and conversion into network-safe subsets of ASCII (typically ACE). Generally, you should be able to use "sensible" strings in your locale's character set as names, but they may appear mangled or escaped in certain contexts such as network transmission.
- Monotone's transmission and storage forms are otherwise unchanged. Packets and database contents are 7-bit clean ASCII.

The remainder of this section is a precise specification of monotone's internationalization behavior.

General Terms

Character set conversion
: The process of mapping a string of bytes representing wide characters from one encoding to another. Per-file character set conversions are specified by a Lua hook `get_charset_conv` which takes a filename and returns a table of two strings: the first represents the "internal" (database) charset, the second represents the "external" (file system) charset.

Line ending conversion
: The process of converting platform-dependent end-of-line codes (`0x0D`, `0x0A`, or the pair `0x0D 0x0A`) from one convention to another. Per-file line ending conversion is specified by a Lua hook `get_linesep_conv` which takes a filename and returns a table of two strings: the first represents the "internal" (database) line ending convention, the second represents the "external" (file system) line ending convention. each string should be one of the three strings "CR", "LF", or "CRLF".

Note that Line ending conversion is always performed on the internal character set, when both character set and line ending conversion are enabled; this behavior is meant to encourage the use of the monotone's "normal form" (UTF-8, '\n') as an internal form for your source files, when working with multiple external forms. Also note that line ending conversion only works on character encodings with the specific code bytes described above, such as ASCII, ISO-8859x, and UTF-8.

Normal form conversion

Character set and line ending conversions done between a "system form" and a "normal form". The system character set form is inferred from the environment, using the various locale environment variables. The system line ending form can be additionally specialized using the `get_system_linesep` hook. No hooks exist for adjusting the system character set, since the system character set must be known during command-line argument processing, before any Lua hooks are loaded.

Monotone's normal form is the UTF-8 character set and the `0x0A` (LF) line ending form. This form is used in any files monotone needs to read, write, and interpret itself, such as: '`MT/revision`', '`MT/work`', '`MT/options`', '`.mt-attrs`'

LDH

Letters, digits, and hyphen: the set of ASCII bytes `0x2D`, `0x30..0x39`, `0x41..0x5A`, and `0x61..0x7A`.

stringprep

RFC 3454, a general framework for mapping, normalizing, prohibiting and bidirectionality checking for international names prior to use in public network protocols.

nameprep

RFC 3491, a specific profile of stringprep, used for preparing international domain names (IDNs)

punycode

RFC 3492, a "bootstring" encoding of unicode into ASCII.

IDNA

RFC 3490, international domain names for applications, a combination of the above technologies (nameprep, punycoding, limiting to LDH characters) to form a specific "ASCII compatible encoding" (ACE) of unicode, signified by the presence of an "unlikely" ACE prefix string "xn–". IDNA is intended to make it possible to use unicode relatively "safely" over legacy ASCII-based applications. the general picture of an IDNA string is this:

```
{ACE-prefix}{LDH-sanitized(punycode(nameprep(UTF-8-string)))}
```

It is important to understand that IDNA encoding does *not* preserve the input string: it both prohibits a wide variety of possible strings and normalizes non-equal strings to supposedly "equivalent" forms.

By default, monotone does *not* decode IDNA when printing to the console (IDNA names are ASCII, which is a subset of UTF-8, so this normal form conversion can still apply, albeit oddly). this behavior is to protect users against security problems associated with malicious use of "similar-looking" characters. If the hook `display_decoded_idna` returns true, IDNA names are decoded for display.

Filenames

- Filenames are subject to normal form conversion.
- Filenames are subject to an additional normal form stage which adjusts for platform name semantics, for example changing the Windows `0x5C` '\' path separator to `0x2F` '/'. This extra processing is performed by boost::filesystem.
- FIXME: Monotone does not properly handle case insensitivity on windows.
- A filename (in normal form) is constrained to be a nonempty sequence of path components, separated by byte `0x2F` (ASCII /), and without a leading or trailing `0x2F`.
- A path component is a nonempty sequence of any UTF-8 character codes except the path separator byte `0x2F` and any ASCII "control codes" (`0x00..0x1F` and `0x7F`).
- The path components "." and ".." are prohibited.
- Manifests and revisions are constructed from the normal form (UTF-8). The LC_COLLATE locale category is *not* used to sort manifest or revision entries.

File contents

- Files are subject to character set conversion and line ending conversion.
- File SHA1 values are calculated from the internal form of the conversions. If the external form of a file differs from the internal form, running a 3rd party program such as `sha1sum` will produce different results than those entries shown in a corresponding manifest.

UI messages

UI messages are displayed via calls to `gettext()`.

Host names

Host names are read on the command-line and subject to normal form conversion. Host names are then split at `0x2E` (ASCII '.'), each component is subject to IDNA encoding, and the components are rejoined.

After processing, host names are stored internally as ASCII. The invariant is that a host name inside monotone contains only sequences of LDH separated by `0x2E`.

Cert names

Read on the command line and subject to normal form conversion and IDNA encoding as a single component. The invariant is that a cert name inside monotone is a single LDH ASCII string.

Cert values

Cert values may be either text or binary, depending on the return value of the hook `cert_is_binary`. If binary, the cert value is never printed to the screen (the literal string "<binary>" is displayed, instead), and is never subjected to line ending or character conversion. If text, the cert value is subject to normal form conversion, as well as having all UTF-8 codes corresponding to ASCII control codes (`0x0..0x1F` and `0x7F`) prohibited in the normal form, except `0x0A` (ASCII LF).

Var domains

Read on the command line and subject to normal form conversion and IDNA encoding as a single component. The invariant is that a var domain inside monotone is a single LDH ASCII string.

Var names and values

Var names and values are assumed to be text, and subject to normal form conversion.

Key names

Read on the command line and subject to normal form conversion and IDNA encoding as an email address (split and joined at '.' and '`@`' characters). The invariant is that a key name inside monotone contains only LDH, `0x2E` (ASCII '.') and `0x40` (ASCII '`@`') characters.

Packets

Packets are 7-bit ASCII. The characters permitted in packets are the union of these character sets:

- The 65 characters of base64 encoding (64 coding `+ "="` pad).
- The 16 characters of hex encoding.
- LDH, '`@`' and '.' characters, as required for key and cert names.
- '[' and ']', the packet delimiters.
- ASCII codes 0x0D (CR), 0x0A (LF), 0x09 (HT), and 0x20 (SP).

The '`.mt-attrs`' file

Now uses 0x0A (ASCII LF) as a delimiter, to permit 0x20 in filenames. This may change in the future.

7.2 Hash Integrity

Some proponents of a competing, proprietary version control system have suggested, in a usenix paper, that the use of a cryptographic hash function such as SHA1 as an identifier for a version is unacceptably unsafe. This section addresses the argument presented in that paper and describes monotone's additional precautions.

To summarize our position:

- the analysis in the paper is wrong,
- even if it were right, monotone is sufficiently safe.

The analysis is wrong

The paper displays a fundamental lack of understanding about what a *cryptographic* hash function is, and how it differs from a normal hash function. Furthermore it confuses accidental collision with attack scenarios, and mixes up its analysis of the risk involved in each. We will try to untangle these issues here.

A cryptographic hash function such as SHA1 is more than just a uniform spread of inputs to an output range. Rather, it must be designed to withstand attempts at:

- reversal: deriving an input value from the output
- collision: finding two different inputs which hash to the same output

Collision is the problem the paper is concerned with. Formally, an n-bit cryptographic hash should cost 2^n work units to collide against a given value, and $sqrt(2^n)$ tries to find a random pair of colliding values. This latter probability is sometimes called the hash's “birthday paradox probability”.

Accidental collision

One way of measuring these bounds is by measuring how single-bit changes in the input affect bits in the hash output. The SHA1 hash has a strong *avalanche property*, which means that flipping *any one bit* in the input will cause on average half the 160 bits in the output code to change. The fanciful VAL1 hash presented in the paper does not have such a property — flipping its first bit when all the rest are zero causes *no change* to any of the 160 output bits — and is completely unsuited for use as a *cryptographic hash*, regardless of the general shape of its probability distribution.

The paper also suggests that birthday paradox probability cannot be used to measure the chance of accidental SHA1 collision on “real inputs”, because birthday paradox probability assumes a uniformly random sample and “real inputs” are not uniformly random. The paper is wrong: the inputs to SHA1 are not what is being measured (and in any case can be arbitrarily long); the collision probability being measured is of *output space*. On output space, the hash is designed to produce uniformly random spread, even given nearly identical inputs. In other words, it is *a primary design criterion* of such a hash that a birthday paradox probability is a valid approximation of its collision probability.

The paper's characterization of risk when hashing “non-random inputs” is similarly deceptive. It presents a fanciful case of a program which is *storing* every possible 2kb block in a file system addressed by SHA1 (the program is trying to find a SHA1 collision). While this scenario *will* very likely encounter a collision *somewhere* in the course of storing all such blocks, the paper neglects to mention that we only expect it to collide after storing about 2^{80} of the 2^{16384} possible such blocks (not to mention the requirements for compute time to search, or disk space to store 2^{80} 2kb blocks).

Noting that monotone can only store 2^{41} bytes in a database, and thus probably some lower number (say 2^{32} or so) active rows, we consider such birthday paradox probability well out of practical sight. Perhaps it will be a serious concern when multi-yottabyte hard disks are common.

Collision attacks

Setting aside accidental collisions, then, the paper's underlying theme of vulnerability rests on the assertion that someone will break SHA1. Breaking a cryptographic hash usually means finding a way to collide it trivially. While we note that SHA1 has in fact resisted attempts at breaking for 8 years already, we cannot say that it will last forever. Someone might break it. We can say, however, that finding a way to trivially collide it only changes the resistance to *active attack*, rather than the behavior of the hash on benign inputs.

Therefore the vulnerability is not that the hash might suddenly cease to address benign blocks well, but merely that additional security precautions might become a requirement to ensure that blocks are benign, rather than malicious. The paper fails to make this distinction, suggesting that a hash becomes "unusable" when it is broken. This is plainly not true, as a number of systems continue to get useful low collision hashing behavior — just not good security behavior — out of "broken" cryptographic hashes such as MD4.

Monotone is probably safe anyways

Perhaps our arguments above are unconvincing, or perhaps you are the sort of person who thinks that practice never lines up with theory. Fair enough. Below we present *practical* procedures you can follow to compensate for the supposed threats presented in the paper.

Collision attacks

A successful collision attack on SHA1, as mentioned, does not disrupt the *probability* features of SHA1 on benign blocks. So if, at any time, you believe SHA1 is "broken", it does *not* mean that you cannot use it for your work with monotone. It means, rather, that you cannot base your *trust* on SHA1 values anymore. You must trust who you communicate with.

The way around this is reasonably simple: if you do not trust SHA1 to prevent malicious blocks from slipping into your communications, you can always augment it by enclosing your communications in more security, such as tunnels or additional signatures on your email posts. If you choose to do this, you will still have the benefit of self-identifying blocks, you will simply cease to trust such blocks unless they come with additional authentication information.

If in the future SHA1 (or, indeed, RSA) becomes accepted as broken we will naturally upgrade monotone to a newer hash or public key scheme, and provide migration commands to recalculate existing databases based on the new algorithm.

Similarly, if you do not trust our vigilance in keeping up to date with cryptography literature, you can modify monotone to use any stronger hash you like, at the cost of isolating your own communications to a group using the modified version. Monotone is free software, and runs atop `crypto++`, so it is both legal and relatively simple to change it to use some other algorithm.

7.3 Rebuilding ancestry

As described in Section 1.3 [Historical records], page 8, monotone revisions contain the SHA1 hashes of their predecessors, which in turn contain the SHA1 hashes of *their* predecessors, and so on until the beginning of history. This means that it is *mathematically impossible* to modify the history of a revision, without some way to defeat SHA1. This is generally a good thing; having immutable history is the point of a version control system, after all, and it turns out to be very important to building a *distributed* version control system like monotone.

It does have one unfortunate consequence, though. It means that in the rare occasion where one *needs* to change a historical revision, it will change the SHA1 of that revision, which will change the text of its children, which will change their SHA1s, and so on; basically the entire history graph will diverge from that point (invalidating all certs in the process).

In practice there are two situations where this might be necessary:

- bugs: monotone has occasionally allowed nonsense, uninterpretable changesets to be generated and stored in the database, and this was not detected until further work had been based off of them.
- advances in crypto: if or when SHA1 is broken, we will need to migrate to a different secure hash.

Obviously, we hope neither of these things will happen, and we've taken lots of precautions against the first recurring; but it is better to be prepared.

If either of these events occur, we will provide migration commands and explain how to use them for the situation in question; this much is necessarily somewhat unpredictable. In the past we've used the `db rebuild` command, which extracts the ancestry graph from the database and then recreates revisions from the manifests only — this preserves the contents of each snapshot, but breaks tracking of file identity across renames — and then reissues all existing certs that you trust, signed with your key.[1]

While the `db rebuild` command can reconstruct the ancestry graph in *your* database, there are practical problems which arise when working in a distributed work group. For example, suppose our group consists of the fictional developers Jim and Beth, and they need to rebuild their ancestry graph. Jim performs a rebuild, and sends Beth an email telling her that he has done so, but the email gets caught by Beth's spam filter, she doesn't see it, and she blithely syncs her database with Jim's. This creates a problem: Jim and Beth have combined the pre-rebuild and post-rebuild databases. Their databases now contain two complete, parallel (but possibly overlapping) copies of their project's ancestry. The "bad" old revisions that they were trying to get rid of are still there, mixed up with the "good" new revisions.

To prevent such messy situations, monotone keeps a table of branch *epochs* in each database. An epoch is just a large bit string associated with a branch. Initially each branch's epoch is zero. Most monotone commands ignore epochs; they are relevant in only two circumstances:

[1] Regardless of who originally signed the certs, after the rebuild they will be signed by you. This means you should be somewhat careful when rebuilding, but it is unavoidable — if you could sign with other people's keys, that would be a rather serious security problem!

- When monotone rebuilds ancestry, it generates a new *random* epoch for each branch in the database.
- When monotone runs netsync between databases, it checks to make sure that all branches involved in the synchronization have the same epochs. If any epochs differ, the netsync is aborted with no changes made to either database. If either side is seeing a branch for the first time, it adopts the epoch of the other side.

Thus, when a user rebuilds their ancestry graph, they select a new epoch and thus effectively disassociate with the group of colleagues they had previously been communicating with. Other members of that group can then decide whether to follow the rebuild user into a new group — by pulling the newly rebuilt ancestry — or to remain behind in the old group.

In our example, if Jim and Beth have epochs, Jim's rebuild creates a new epoch for their branch, in his database. This causes monotone to reject netsync operations between Jim and Beth; it doesn't matter if Beth loses Jim's email. When she tries to synchronize with him, she receives an error message indicating that the epoch does not match. She must then discuss the matter with Jim and settle on a new course of action — probably pulling Jim's database into a fresh database on Beth's end – before future synchronizations will succeed.

Best practices

The previous section described the theory and rationale behind rebuilds and epochs. Here we discuss the practical consequences of that discussion.

If you decide you must rebuild your ancestry graph — generally by announcement of a bug from the monotone developers — the first thing to do is get everyone to sync their changes with the central server; if people have unshared changes when the database is rebuilt, they will have trouble sharing them afterwards.

Next, the project should pick a designated person to take down the netsync server, rebuild their database, and put the server back up with the rebuilt ancestry in it. Everybody else should then pull this history into a fresh database, check out again from this database, and continue working as normal.

In complicated situations, where people have private branches, or ancestries cross organizational boundaries, matters are more complex. The basic approach is to do a local rebuild, then after carefully examining the new revision IDs to convince yourself that the rebuilt graph is the same as the upstream subgraph, use the special `db epoch` commands to force your local epochs to match the upstream ones. (You may also want to do some fiddling with certs, to avoid getting duplicate copies of all of them; if this situation ever arises in real life we'll figure out how exactly that should work.) Be very careful when doing this; you're explicitly telling monotone to let you shoot yourself in the foot, and it will let you.

Fortunately, this process should be extremely rare; with luck, it will never happen at all. But this way we're prepared.

8 Man Page

8.1 NAME

monotone – distributed version control system

8.2 SYNOPSIS

monotone *[options]* <*command*> *[parameters]*

Options, which affect global behavior or set default values, come first in the argument list. A single command must follow, indicating the operation to perform, followed by parameters which vary depending on the command.

8.2.1 Note

This man page is a summary of some of the features and commands of **monotone**, but it is not the most detailed source of information available. For a complete discussion of the concepts and a tutorial on its use, please refer to the texinfo manual (via the **info monotone** command, or online).

8.2.2 Commands

comment `<id>`
: Write a comment cert for a revision.

approve `<id>`
: Make a "branch" cert approving of a revision's membership in a branch.

disapprove `<id>`
: Disapprove of a revision, committing the inverse changes as as a descendent of the disapproved revision.

tag `<id> <tagname>`
: Put a symbolic tag cert on a revision.

testresult `<id> (0|1|true|false|yes|no|pass|fail)`
: Indicate a passing or failing test result on a revision.

diff `[--revision=<id1> [--revision=<id2>] ] [<pathname>...]`
: Show diffs between working copy and database.

status `[<pathname>...]`
: Show status of working copy.

log `[id]`
: Show historical log of revisions, starting from working copy base revision, or *[id]* if given.

cert `<id> <certname> [certval]`
: Create a custom cert for a revision. Reads cert value from stdin if no value given on command line.

genkey `<keyid>`
: Generate an RSA key-pair and store it in the database.

dropkey *<keyid>*
: Drop a public and/or private key.

chkeypass *<keyid>*
: Change passphrase of the private half of a key.

list certs *<id>*
ls certs *<id>*
: List certs associated with revision.

list keys *[partial-id]*
ls keys *[partial-id]*
: List keys matching glob, or list all keys if no glob given.

list branches
ls branches
: List all branches.

list tags
ls tags List all tags.

list known *[<pathname>...]*
ls known *[<pathname>...]*
: List files which are in revision's manifest, or are on the work list of the working copy.

list unknown *[<pathname>...]*
ls unknown *[<pathname>...]*
: List files in working copy, but not in revision's manifest or work list.

list ignored *[<pathname>...]*
ls ignored *[<pathname>...]*
: List files intentionally ignored due to the ignore_file hook.

list missing *[<pathname>...]*
ls missing *[<pathname>...]*
: List files in revision's manifest, but not in working copy.

fdata *<id>*
: Write file data packet to stdout.

fdelta *<oldid> <newid>*
: Write file delta packet to stdout.

mdata *<id>*
: Write manifest data packet to stdout.

mdelta *<oldid> <newid>*
: Write manifest delta packet to stdout.

rcerts *<id>*
: Write revision cert packets to stdout.

rdata *<id>*
: Write revision data packet to stdout.

privkey `<id>`
: Write private key packet to stdout.

pubkey `<id>`
: Write public key packet to stdout.

read *`[<file1> [<file2> [...]]]`*
: Read packets from files or stdin.

cvs_import `<cvsroot>/<module>`
: Import all versions in a CVS module. Reconstructs revisions and converts metadata to certificates. A private signing key must already exist in the database.

rcs_import `<rcsfile> ...`
: Import all file versions in RCS files. Does not reconstruct revisions across the entire tree. You do not want this command, it is for debugging; use cvs_import.

checkout *`[--revision=revision-id] [<directory>]`*
co *`[--revision=revision-id] [<directory>]`*
: Check out revision from database, into directory.

cat *`(file <id>|manifest [<id>]|revision [<id>])`*
: Write file, manifest or revision from database to stdout.

heads Show unmerged heads of branch, or report when branch is merged.

merge Merge unmerged heads of branch.

add `<pathname>` *`[...]`*
: Add files to working copy. adding a file does not copy it into the database, merely adds it to the work list. You must **commit** your changes in order to copy added files to the database.

drop `<pathname>` *`[...]`*
: Drop files from working copy. Files are not deleted from working copy, merely noted as removals in the work list.

rename `<src> <dst>`
: Rename files from <*src*> to <*dst*> in working copy.

commit *`[--message=log message | --message-file=log message file]`*
`[<pathname>...]`
: Commit working copy to database. Each commit has a changelog message associated with it. If '`--message`' is provided on the command line, it is used; if '`--message-file`' is provided, the content of the named file will be used as a commit message. If the filename is '-' the commit message will be read from standard input. Otherwise a log message editor will be invoked. If the file '`MT/log`' exists and is non-empty, its content is used to prefill the editor. You cannot specify both '`--message`' and '`--message-file`' at the same time, and if '`MT/log`' exists and is non-empty, you can cannot specify either of them at all.

update *`[--revision=revision]`*
: Update working copy.

refresh_inodeprints
: Turn on inodeprints mode, and force a cache refresh.

push `<host>` `<glob>`
: Push contents of branches matching <*glob*> to database on <*host*>

pull `<host>` `<glob>`
: Pull contents of branches matching <*glob*> from database on <*host*>

sync `<host>` `<glob>`
: Sync contents of branches matching <*glob*> with database on <*host*>

serve `<host>` `<glob>`
: Serve contents of branches matching <*glob*> at network address <*host*>

automate *`(interface_version|heads|ancestors|attributes|parents|descendents|children|graph|erase_ancestors|toposort|ancestry_difference|leaves|inventory|stdio|certs|select)`*
: Scripting interface.

db *`(init|info|version|dump|load|migrate|execute <sql>)`*
: Manipulate database state.

8.3 DESCRIPTION

Monotone is a version control system, which allows you to keep old versions of files, as well as special *revisions* and *manifests* which describe the edit history, location, and content of files in a tree. Unlike other systems, versions in monotone are *identified* by cryptographic hash, and operations are authenticated by individual users' evaluating cryptographic signatures on meta-data, rather than any central authority.

Monotone keeps a collection of versions in a single-file relational database. It is essentially serverless, using network servers only as untrusted communication facilities. A monotone database is a regular file, which contains all the information needed to extract previous versions of files, verify signatures, merge and modify versions, and communicate with network servers.

8.4 OPTIONS

–help
: Print help message.

–debug
: Turn on debugging log on standard error stream. This is very verbose. Default is to be silent, unless an error occurs, in which case failure log is dumped.

–quiet
: Turn off normal progress messages.

–dump=`<file>`
: Dump debugging log to <*file*> on failure.

–nostd
: Do not evaluate `"standard"` Lua hooks compiled into **monotone**.

–norc
: Do not load Lua hooks from user's `~`**/.monotone/monotonerc** or the working copy's **MT/monotonerc** file.

–rcfile=`<file>`
: Load extra Lua hooks from *file* (may be given multiple times).

–db=<*file*>
: Use database in *file*.

–key=<*keyid*>
-k <*keyid*>
: Use *keyid* for operations which produce RSA signatures. Default is inferred from presence of unique private key in database. Can also be customized on a per-branch basis with hook function **get_branch_key(branchname)**.

–branch=<*branchname*>
-b <*branchname*>
: Use *branchname* for operations on a branch. Default is inferred in operations on existing branches (commit, update, etc).

–ticker=[dot|count]
: Use the given method to print tickers. The **count** method prints the count for each ticker on one line, incrementing the numbers in place, while the **dot** method prints a continuous string of characters (like some programs provide a progress line of dots). The default is **count**.

–revision=<*id*>
-r <*id*>
: Use the given revision with the current command. The revision id could be specified as a selector. Section 3.1 [Selectors], page 44.

–message-file=<*log message file*>
: Use the content of the given file as the changelog when committing a new revision rather than invoking the log message editor. If the filename is '-' the changelog is read from standard input. Currently this option only applies to the commit command, but it may also apply to the comment command in the future.

–message=<*log message*>
-m <*log message*>
: Use the given message as the changelog when committing a new revision rather than invoking the log message editor. Currently this option only applies to the commit command but it may also apply to the comment command in the future.

–author=<*author email*>
: Use the given author as the value of the `author` cert when committing a new revision, rather than the default author. Useful when committing a patch on behalf of someone else, or when importing history from another version control system.

–date=<*date and time*>
: Use the given given date and time as value of the `date` cert when committing a new revision, rather than the current time. Useful when importing history from another version control system.

–root=<*root dir*>
: Stop the search for a working copy (containing the '`MT`' directory) at the specified root directory rather than at the physical root of the filesystem.

–xargs=<*file*>
: Inject the contents of the file in place among the command line arguments. This may be useful in case the command line would otherwise become too long for your system. This option can be used more than once if needed.

-@ <*file*> An alias for **–xargs=<*file*>**.

8.5 ENVIRONMENT

EDITOR Used to edit comments, log messages, etc.

VISUAL Used in preference to **EDITOR**, if set.

8.6 FILES

$HOME/.monotone/monotonerc
: A Lua script, used as a customization file.

8.7 NOTES

Command line options override environment variables and settings in Lua scripts (such as **monotonerc**)

8.8 SEE ALSO

info monotone

8.9 BUGS

see http://savannah.nongnu.org/bugs/?group=monotone

8.10 AUTHOR

graydon hoare <graydon@pobox.com>

Appendix A Default hooks

This section contains the entire source code of the standard hook file, that is built in to the monotone executable, and read before any user hooks files (unless '`--nostd`' is passed). It contains the default values for all hooks.

```
-- this is the standard set of lua hooks for monotone;
-- user-provided files can override it or add to it.

function temp_file()
   local tdir
   tdir = os.getenv("TMPDIR")
   if tdir == nil then tdir = os.getenv("TMP") end
   if tdir == nil then tdir = os.getenv("TEMP") end
   if tdir == nil then tdir = "/tmp" end
   return mkstemp(string.format("%s/mt.XXXXXX", tdir))
end

function execute(path, ...)
   local pid
   local ret = -1
   pid = spawn(path, unpack(arg))
   if (pid ~= -1) then ret, pid = wait(pid) end
   return ret
end

-- Wrapper around execute to let user confirm in the case where a subprocess
-- returns immediately
-- This is needed to work around some brokenness with some merge tools
-- (e.g. on OS X)
function execute_confirm(path, ...)
   execute(path, unpack(arg))
   print(gettext("Press enter when the subprocess has completed"))
   io.read()
   return ret
end

-- attributes are persistent metadata about files (such as execute
-- bit, ACLs, various special flags) which we want to have set and
-- re-set any time the files are modified. the attributes themselves
-- are stored in a file .mt-attrs, in the working copy (and
-- manifest). each (f,k,v) triple in an attribute file turns into a
-- call to attr_functions[k](f,v) in lua.

if (attr_init_functions == nil) then
   attr_init_functions = {}
end
```

```
attr_init_functions["execute"] =
   function(filename)
      if (is_executable(filename)) then
        return "true"
      else
        return nil
      end
   end

attr_init_functions["manual_merge"] =
   function(filename)
      if (binary_file(filename)) then
        return "true" -- binary files must merged manually
      else
        return nil
      end
   end

if (attr_functions == nil) then
   attr_functions = {}
end

attr_functions["execute"] =
   function(filename, value)
      if (value == "true") then
         make_executable(filename)
      end
   end


function ignore_file(name)
   -- project specific
   if (ignored_files == nil) then
      ignored_files = {}
      local ignfile = io.open(".mt-ignore", "r")
      if (ignfile ~= nil) then
         local line = ignfile:read()
         while (line ~= nil)
         do
            table.insert(ignored_files, line)
            line = ignfile:read()
         end
         io.close(ignfile)
      end
   end
   for i, line in pairs(ignored_files)
```

```
do
   if (regex.search(line, name)) then return true end
end
-- c/c++
if (string.find(name, "%.a$")) then return true end
if (string.find(name, "%.so$")) then return true end
if (string.find(name, "%.o$")) then return true end
if (string.find(name, "%.la$")) then return true end
if (string.find(name, "%.lo$")) then return true end
if (string.find(name, "^core$")) then return true end
if (string.find(name, "/core$")) then return true end
-- python
if (string.find(name, "%.pyc$")) then return true end
if (string.find(name, "%.pyo$")) then return true end
-- TeX
if (string.find(name, "%.aux$")) then return true end
-- backup files
if (string.find(name, "%.bak$")) then return true end
if (string.find(name, "%.orig$")) then return true end
if (string.find(name, "%.rej$")) then return true end
if (string.find(name, "%~$")) then return true end
-- editor temp files
-- vim creates .foo.swp files
if (string.find(name, "%.[^/]*%.swp$")) then return true end
-- emacs creates #foo# files
if (string.find(name, "%#[^/]*%#$")) then return true end
-- autotools detritus:
if (string.find(name, "^autom4te.cache/")) then return true end
if (string.find(name, "/autom4te.cache/")) then return true end
if (string.find(name, "^.deps/")) then return true end
if (string.find(name, "/.deps/")) then return true end
-- Cons/SCons detritus:
if (string.find(name, "^.consign$")) then return true end
if (string.find(name, "/.consign$")) then return true end
if (string.find(name, "^.sconsign$")) then return true end
if (string.find(name, "/.sconsign$")) then return true end
-- other VCSes:
if (string.find(name, "^CVS/")) then return true end
if (string.find(name, "/CVS/")) then return true end
if (string.find(name, "^%.svn/")) then return true end
if (string.find(name, "/%.svn/")) then return true end
if (string.find(name, "^SCCS/")) then return true end
if (string.find(name, "/SCCS/")) then return true end
if (string.find(name, "^_darcs/")) then return true end
if (string.find(name, "^.cdv/")) then return true end
if (string.find(name, "^.git/")) then return true end
if (string.find(name, "%.scc$")) then return true end
```

```
   -- desktop/directory configuration metadata
   if (string.find(name, "^.DS_Store$")) then return true end
   if (string.find(name, "/.DS_Store$")) then return true end
   if (string.find(name, "^desktop.ini$")) then return true end
   if (string.find(name, "/desktop.ini$")) then return true end
   return false;
end

-- return true means "binary", false means "text",
-- nil means "unknown, try to guess"
function binary_file(name)
   local lowname=string.lower(name)
   -- some known binaries, return true
   if (string.find(lowname, "%.gif$")) then return true end
   if (string.find(lowname, "%.jpe?g$")) then return true end
   if (string.find(lowname, "%.png$")) then return true end
   if (string.find(lowname, "%.bz2$")) then return true end
   if (string.find(lowname, "%.gz$")) then return true end
   if (string.find(lowname, "%.zip$")) then return true end
   -- some known text, return false
   if (string.find(lowname, "%.cc?$")) then return false end
   if (string.find(lowname, "%.cxx$")) then return false end
   if (string.find(lowname, "%.hh?$")) then return false end
   if (string.find(lowname, "%.hxx$")) then return false end
   if (string.find(lowname, "%.lua$")) then return false end
   if (string.find(lowname, "%.texi$")) then return false end
   if (string.find(lowname, "%.sql$")) then return false end
   -- unknown - read file and use the guess-binary
   -- monotone built-in function
   return guess_binary_file_contents(name)
end

function edit_comment(basetext, user_log_message)
   local exe = nil
   if (program_exists_in_path("vi")) then exe = "vi" end
   if (program_exists_in_path("notepad.exe")) then exe = "notepad.exe" end
   local visual = os.getenv("VISUAL")
   if (visual ~= nil) then exe = visual end
   local editor = os.getenv("EDITOR")
   if (editor ~= nil) then exe = editor end

   if (exe == nil) then
      io.write("Could not find editor to enter commit message\n"
               .. "Try setting the environment variable EDITOR\n")
      return nil
   end
   local tmp, tname = temp_file()
   if (tmp == nil) then return nil end
   basetext = "MT: " .. string.gsub(basetext, "\n", "\nMT: ") .. "\n"
   tmp:write(user_log_message)
   tmp:write(basetext)
   io.close(tmp)

   if (execute(exe, tname) ~= 0) then
      io.write(string.format(gettext("Error running editor ’%s’ to enter log message\n"),
                             exe))
      os.remove(tname)
      return nil
   end

   tmp = io.open(tname, "r")
   if (tmp == nil) then os.remove(tname); return nil end
   local res = ""
   local line = tmp:read()
   while(line ~= nil) do
      if (not string.find(line, "^MT:")) then
         res = res .. line .. "\n"
      end
      line = tmp:read()
   end
   io.close(tmp)
   os.remove(tname)
   return res
end


function non_blocking_rng_ok()
   return true
end


function persist_phrase_ok()
   return true
end

-- trust evaluation hooks

function intersection(a,b)
   local s={}
   local t={}
   for k,v in pairs(a) do s[v] = 1 end
   for k,v in pairs(b) do if s[v] ~= nil then table.insert(t,v) end end
   return t
```

```
end

function get_revision_cert_trust(signers, id, name, val)
   return true
end

function get_manifest_cert_trust(signers, id, name, val)
   return true
end

function get_file_cert_trust(signers, id, name, val)
   return true
end

function accept_testresult_change(old_results, new_results)
   local reqfile = io.open("MT/wanted-testresults", "r")
   if (reqfile == nil) then return true end
   local line = reqfile:read()
   local required = {}
   while (line ~= nil)
   do
      required[line] = true
      line = reqfile:read()
   end
   io.close(reqfile)
   for test, res in pairs(required)
   do
      if old_results[test] == true and new_results[test] ~= true
      then
         return false
      end
   end
   return true
end

-- merger support

function merge2_meld_cmd(lfile, rfile)
   return
   function()
      return execute("meld", lfile, rfile)
   end
end

function merge3_meld_cmd(lfile, afile, rfile)
   return
   function()
```

```
      return execute("meld", lfile, afile, rfile)
   end
end

function merge2_tortoise_cmd(lfile, rfile, outfile)
   return
   function()
      return execute("tortoisemerge",
                     string.format("/theirs:%s", lfile),
                     string.format("/mine:%s", rfile),
                     string.format("/merged:%s", outfile))
   end
end

function merge3_tortoise_cmd(lfile, afile, rfile, outfile)
   return
   function()
      return execute("tortoisemerge",
                     string.format("/base:%s", afile),
                     string.format("/theirs:%s", lfile),
                     string.format("/mine:%s", rfile),
                     string.format("/merged:%s", outfile))
   end
end

function merge2_vim_cmd(vim, lfile, rfile, outfile)
   return
   function()
      return execute(vim, "-f", "-d", "-c", string.format("file %s", outfile),█
                     lfile, rfile)
   end
end

function merge3_vim_cmd(vim, afile, lfile, rfile, outfile)
   return
   function()
      return execute(vim, "-f", "-d", "-c", string.format("file %s", outfile),█
                     afile, lfile, rfile)
   end
end

function merge3_rcsmerge_vim_cmd(merge, vim, lfile, afile, rfile, outfile)
   return
   function()
      -- XXX: This is tough - should we check if conflict markers stay or not?█
      -- If so, we should certainly give the user some way to still force
      -- the merge to proceed since they can appear in the files (and I saw
      -- that). --pasky
      if execute(merge, lfile, afile, rfile) == 0 then
         copy_text_file(lfile, outfile);
         return 0
      end
      return execute(vim, "-f", "-c", string.format("file %s", outfile),
                     lfile)
   end
end

function merge2_emacs_cmd(emacs, lfile, rfile, outfile)
   local elisp = "(ediff-merge-files \"%s\" \"%s\" nil \"%s\")"
   return
   function()
      return execute(emacs, "-eval",
                     string.format(elisp, lfile, rfile, outfile))
   end
end

function merge3_emacs_cmd(emacs, lfile, afile, rfile, outfile)
   local elisp = "(ediff-merge-files-with-ancestor \"%s\" \"%s\" \"%s\" nil \"%s\")"
   local cmd_fmt = "%s -eval " .. elisp
   return
   function()
      execute(emacs, "-eval",
              string.format(elisp, lfile, rfile, afile, outfile))
   end
end

function merge2_xxdiff_cmd(left_path, right_path, merged_path, lfile, rfile, outfile)
   return
   function()
      return execute("xxdiff",
                     "--title1", left_path,
                     "--title2", right_path,
                     lfile, rfile,
                     "--merged-filename", outfile)
   end
end

function merge3_xxdiff_cmd(left_path, anc_path, right_path, merged_path,
                           lfile, afile, rfile, outfile)
   return
   function()
      return execute("xxdiff",
                     "--title1", left_path,
                     "--title2", right_path,
                     "--title3", merged_path,
                     lfile, afile, rfile,
                     "--merge",
                     "--merged-filename", outfile)
   end
end

function merge2_kdiff3_cmd(left_path, right_path, merged_path, lfile, rfile, outfile)
   return
   function()
      return execute("kdiff3",
                     "--L1", left_path,
                     "--L2", right_path,
                     lfile, rfile,
                     "-o", outfile)
   end
end

function merge3_kdiff3_cmd(left_path, anc_path, right_path, merged_path,
                           lfile, afile, rfile, outfile)
   return
   function()
      return execute("kdiff3",
                     "--L1", anc_path,
                     "--L2", left_path,
                     "--L3", right_path,
                     afile, lfile, rfile,
                     "--merge",
                     "--o", outfile)
   end
end

function merge2_opendiff_cmd(left_path, right_path, merged_path, lfile, rfile, outfile)
   return
   function()
      -- As opendiff immediately returns, let user confirm manually
      return execute_confirm("opendiff",lfile,rfile,"-merge",outfile)
  end
end

function merge3_opendiff_cmd(left_path, anc_path, right_path, merged_path, lfile, afile, rf
   return
   function()
      -- As opendiff immediately returns, let user confirm manually
      execute_confirm("opendiff",lfile,rfile,"-ancestor",afile,"-merge",outfile)
   end
end
```

```
function write_to_temporary_file(data)
   tmp, filename = temp_file()
   if (tmp == nil) then
      return nil
   end;
   tmp:write(data)
   io.close(tmp)
   return filename
end

function copy_text_file(srcname, destname)
   src = io.open(srcname, "r")
   if (src == nil) then return nil end
   dest = io.open(destname, "w")
   if (dest == nil) then return nil end

   while true do
      local line = src:read()
      if line == nil then break end
      dest:write(line, "\n")
   end

   io.close(dest)
   io.close(src)
end

function read_contents_of_file(filename, mode)
   tmp = io.open(filename, mode)
   if (tmp == nil) then
      return nil
   end
   local data = tmp:read("*a")
   io.close(tmp)
   return data
end

function program_exists_in_path(program)
   return existsonpath(program) == 0
end

function get_preferred_merge2_command (tbl)
   local cmd = nil
   local left_path = tbl.left_path
   local right_path = tbl.right_path
   local merged_path = tbl.merged_path
   local lfile = tbl.lfile
   local rfile = tbl.rfile
   local outfile = tbl.outfile

   local editor = os.getenv("EDITOR")
   if editor ~= nil then editor = string.lower(editor) else editor = "" end


   if program_exists_in_path("kdiff3") then
      cmd =   merge2_kdiff3_cmd (left_path, right_path, merged_path, lfile, rfile, outfile)
   elseif program_exists_in_path ("xxdiff") then
      cmd = merge2_xxdiff_cmd (left_path, right_path, merged_path, lfile, rfile, outfile)
   elseif program_exists_in_path ("opendiff") then
      cmd = merge2_opendiff_cmd (left_path, right_path, merged_path, lfile, rfile, outfile)
   elseif program_exists_in_path ("TortoiseMerge") then
      cmd = merge2_tortoise_cmd(lfile, rfile, outfile)
   elseif string.find(editor, "emacs") ~= nil or string.find(editor, "gnu") ~= nil then
      if string.find(editor, "xemacs") and program_exists_in_path("xemacs") then
         cmd = merge2_emacs_cmd ("xemacs", lfile, rfile, outfile)
      elseif program_exists_in_path("emacs") then
         cmd = merge2_emacs_cmd ("emacs", lfile, rfile, outfile)
      end
   elseif string.find(editor, "vim") ~= nil then
      io.write (string.format("\nWARNING: 'vim' was choosen to perform external 2-way merge
          "You should merge all changes to *LEFT* file due to limitation of program\n"..
          "arguments.\n\n"))
      if os.getenv ("DISPLAY") ~= nil and program_exists_in_path ("gvim") then
         cmd = merge2_vim_cmd ("gvim", lfile, rfile, outfile)
      elseif program_exists_in_path ("vim") then
         cmd = merge2_vim_cmd ("vim", lfile, rfile, outfile)
      end
   elseif program_exists_in_path ("meld") then
      tbl.meld_exists = true
      io.write (string.format("\nWARNING: 'meld' was choosen to perform external 2-way merg
          "You should merge all changes to *LEFT* file due to limitation of program\n"..
          "arguments.\n\n"))
      cmd = merge2_meld_cmd (lfile, rfile)
   end
   return cmd
end

function merge2 (left_path, right_path, merged_path, left, right)
   local ret = nil
   local tbl = {}

   tbl.lfile = nil
   tbl.rfile = nil
   tbl.outfile = nil
   tbl.meld_exists = false

   tbl.lfile = write_to_temporary_file (left)
   tbl.rfile = write_to_temporary_file (right)
   tbl.outfile = write_to_temporary_file ("")

   if tbl.lfile ~= nil and tbl.rfile ~= nil and tbl.outfile ~= nil
   then
      tbl.left_path = left_path
      tbl.right_path = right_path
      tbl.merged_path = merged_path

      local cmd = get_preferred_merge2_command (tbl)

      if cmd ~=nil
      then
         io.write (string.format(gettext("executing external 2-way merge command\n")))
         cmd ()
         if tbl.meld_exists
         then
            ret = read_contents_of_file (tbl.lfile, "r")
         else
            ret = read_contents_of_file (tbl.outfile, "r")
         end
         if string.len (ret) == 0
         then
            ret = nil
         end
      else
         io.write (string.format("No external 2-way merge command found.\n"..
            "You may want to check that $EDITOR is set to an editor that supports 2-way merg
            "set this explicitly in your get_preferred_merge2_command hook,\n"..
            "or add a 2-way merge program to your path.\n\n"))
      end
   end

   os.remove (tbl.lfile)
   os.remove (tbl.rfile)
   os.remove (tbl.outfile)

   return ret
end

function get_preferred_merge3_command (tbl)
   local cmd = nil
   local left_path = tbl.left_path
   local anc_path = tbl.anc_path
   local right_path = tbl.right_path
   local merged_path = tbl.merged_path
   local lfile = tbl.lfile
   local afile = tbl.afile
   local rfile = tbl.rfile
   local outfile = tbl.outfile

   local editor = os.getenv("EDITOR")
   if editor ~= nil then editor = string.lower(editor) else editor = "" end

   local merge = os.getenv("MTMERGE")
   -- TODO: Support for rcsmerge_emacs
   if merge ~= nil and string.find(editor, "vim") ~= nil then
      if os.getenv ("DISPLAY") ~= nil and program_exists_in_path ("gvim") then
         cmd = merge3_rcsmerge_vim_cmd (merge, "gvim", lfile, afile, rfile, outfile)
      elseif program_exists_in_path ("vim") then
         cmd = merge3_rcsmerge_vim_cmd (merge, "vim", lfile, afile, rfile, outfile)
      end

   elseif program_exists_in_path("kdiff3") then
      cmd = merge3_kdiff3_cmd (left_path, anc_path, right_path, merged_path, lfile, afile, r
   elseif program_exists_in_path ("xxdiff") then
      cmd = merge3_xxdiff_cmd (left_path, anc_path, right_path, merged_path, lfile, afile, r
   elseif program_exists_in_path ("opendiff") then
      cmd = merge3_opendiff_cmd (left_path, anc_path, right_path, merged_path, lfile, afile
   elseif program_exists_in_path ("TortoiseMerge") then
      cmd = merge3_tortoise_cmd(lfile, afile, rfile, outfile)
   elseif string.find(editor, "emacs") ~= nil or string.find(editor, "gnu") ~= nil then
      if string.find(editor, "xemacs") and program_exists_in_path ("xemacs") then
         cmd = merge3_emacs_cmd ("xemacs", lfile, afile, rfile, outfile)
      elseif program_exists_in_path ("emacs") then
         cmd = merge3_emacs_cmd ("emacs", lfile, afile, rfile, outfile)
      end
   elseif string.find(editor, "vim") ~= nil then
      io.write (string.format("\nWARNING: 'vim' was choosen to perform external 2-way merge
          "You should merge all changes to *LEFT* file due to limitation of program\n"..
          "arguments.  The order of the files is ancestor, left, right.\n\n"))
      if os.getenv ("DISPLAY") ~= nil and program_exists_in_path ("gvim") then
         cmd = merge3_vim_cmd ("gvim", afile, lfile, rfile, outfile)
      elseif program_exists_in_path ("vim") then
         cmd = merge3_vim_cmd ("vim", afile, lfile, rfile, outfile)
      end
   elseif program_exists_in_path ("meld") then
      tbl.meld_exists = true
      io.write (string.format("\nWARNING: 'meld' was choosen to perform external 3-way merg
          "You should merge all changes to *CENTER* file due to limitation of program\n"..
          "arguments.\n\n"))
      cmd = merge3_meld_cmd (lfile, afile, rfile)
   end

   return cmd
end

function merge3 (anc_path, left_path, right_path, merged_path, ancestor, left, right)
   local ret
   local tbl = {}

   tbl.anc_path = anc_path
   tbl.left_path = left_path
   tbl.right_path = right_path

   tbl.merged_path = merged_path
   tbl.afile = nil
   tbl.lfile = nil
   tbl.rfile = nil
   tbl.outfile = nil
   tbl.meld_exists = false
   tbl.lfile = write_to_temporary_file (left)
   tbl.afile =   write_to_temporary_file (ancestor)
   tbl.rfile =   write_to_temporary_file (right)
   tbl.outfile = write_to_temporary_file ("")

   if tbl.lfile ~= nil and tbl.rfile ~= nil and tbl.afile ~= nil and tbl.outfile ~= nil
   then
      local cmd =   get_preferred_merge3_command (tbl)
      if cmd ~=nil
      then
         io.write (string.format(gettext("executing external 3-way merge command\n")))
         cmd ()
         if tbl.meld_exists
         then
            ret = read_contents_of_file (tbl.afile, "r")
         else
            ret = read_contents_of_file (tbl.outfile, "r")
         end
         if string.len (ret) == 0
         then
            ret = nil
         end
      else
         io.write (string.format("No external 3-way merge command found.\n"..
            "You may want to check that $EDITOR is set to an editor that supports 3-way merg
            "set this explicitly in your get_preferred_merge3_command hook,\n"..
            "or add a 3-way merge program to your path.\n\n"))
      end
   end

   os.remove (tbl.lfile)
   os.remove (tbl.rfile)
   os.remove (tbl.afile)
   os.remove (tbl.outfile)

   return ret
end

-- expansion of values used in selector completion

function expand_selector(str)

   -- something which looks like a generic cert pattern
   if string.find(str, "^[^=]*=.*$")
   then
      return ("c:" .. str)
   end

   -- something which looks like an email address
   if string.find(str, "[%w%-_]+@[%w%-_]+")
   then
      return ("a:" .. str)
   end

   -- something which looks like a branch name
   if string.find(str, "[%w%-]+%.[%w%-]+")
   then
      return ("b:" .. str)
   end

   -- a sequence of nothing but hex digits
   if string.find(str, "^%x+$")
   then
      return ("i:" .. str)
   end

   -- tries to expand as a date
   local dtstr = expand_date(str)
   if  dtstr ~= nil
   then
      return ("d:" .. dtstr)
   end

   return nil
```

```
end

-- expansion of a date expression
function expand_date(str)
   -- simple date patterns
   if string.find(str, "^19%d%d%-%d%d")
      or string.find(str, "^20%d%d%-%d%d")
   then
      return (str)
   end

   -- "now"
   if str == "now"
   then
      local t = os.time(os.date('!*t'))
      return os.date("%FT%T", t)
   end

         -- today don't uses the time
   if str == "today"
   then
      local t = os.time(os.date('!*t'))
      return os.date("%F", t)
   end

   -- "yesterday", the source of all hangovers
   if str == "yesterday"
   then
      local t = os.time(os.date('!*t'))
      return os.date("%F", t - 86400)
   end

   -- "CVS style" relative dates such as "3 weeks ago"
   local trans = {
      minute = 60;
      hour = 3600;
      day = 86400;
      week = 604800;
      month = 2678400;
      year = 31536000
   }
   local pos, len, n, type = string.find(str, "(%d+) ([minutehordaywk]+)s? ago")
   if trans[type] ~= nil
   then
      local t = os.time(os.date('!*t'))
      if trans[type] <= 3600
      then
```

```
        return os.date("%FT%T", t - (n * trans[type]))
      else
        return os.date("%F", t - (n * trans[type]))
      end
   end

   return nil
end

function use_inodeprints()
   return false
end

external_diff_default_args = "-u"

-- default external diff, works for gnu diff
function external_diff(file_path, data_old, data_new, is_binary, diff_args, rev_old, rev_ne
   local old_file = write_to_temporary_file(data_old);
   local new_file = write_to_temporary_file(data_new);

   if diff_args == nil then diff_args = external_diff_default_args end
   execute("diff", diff_args, "--label", file_path .. "\told", old_file, "--label", file_pa

   os.remove (old_file);
   os.remove (new_file);
end
```

Index