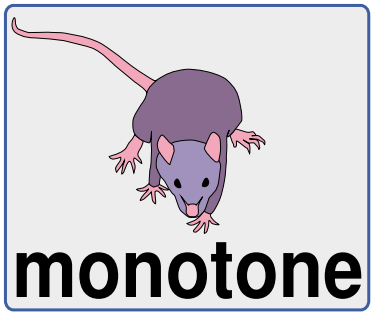
Modern Source Code Management and monotone

Version 0.1

Richard Levitte, LP
mailto:levittelp.se

October 6, 2005



Content

- **The purpose of `monotone` and the consequences**
- **What does a development tree look like?**
- **Workflow, storage and control**
- **In practice**
- **A word on uniqueness and world-wide distribution**
- **Tools**
- **Where to go next**

The purpose of `monotone` and the consequences

It's distributed and works off-line

- Every participant has a complete snapshot of the repository.
- Committing changes and synchronising with remote databases are separate operations.
- Every participant can set up a server of his/her own at any time.
- There's no dependency on a single central server.
- Every file content has a globally unique identifier (using SHA-1).
- Every revision has a globally unique identifier (using SHA-1).
- Several lines of development can exist in parallell within a branch.
- Commit-then-update-after-possibly-merge is encouraged.

The purpose of `monotone` and the consequences

It can be set up and used by anyone

- No external database server. `monotone` uses SQLite.
- No external communication server needed. `monotone` has it's own communication protocol.
- No special privileges needed, apart from the `monotone` port being open.

The purpose of `monotone` and the consequences

It leaves an audit trail

- All changes to the repository are signed cryptographically.

The purpose of `monotone` and the consequences

It's changeset-oriented and atomic

- There are two common views on change history: per-file and per-change.
- `monotone` uses the per-change view.

- All operations that change anything are atomic.
- All operations that change anything are rolled back on error.

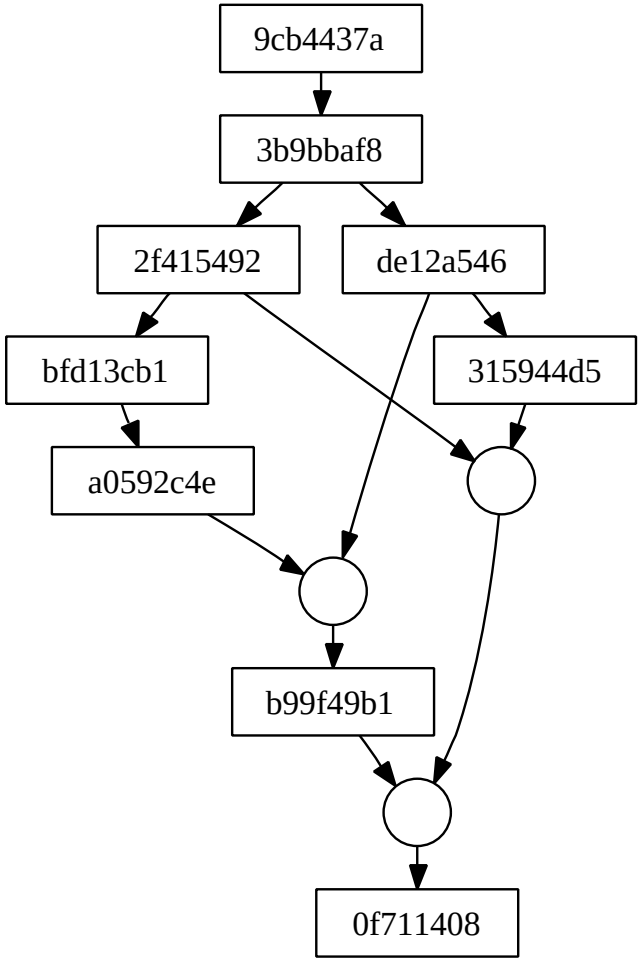
The purpose of `monotone` and the consequences

It's branch-oriented

- Every branch is equal.
- There is no main trunk.
- There is no vendor branch.

The purpose of `monotone` and the consequences

It's history-sensitive



- The history handled by `monotone` is a directed acyclic graph (DAG).
- Every revision contains pointers to it's parents.

The purpose of `monotone` and the consequences

It's quite easy to understand, and it's consistent

- The internal layout and interconnection of revisions is well documented.
- There are no (should not be :-)) corner case.
- Merges consider previous history, so nothing is repeated (i.e. no unnecessary conflicts).

What does a development tree look like?

The revision itself

The revision is information about a change, and the revision ID is it's SHA-1 hash.

```
new_manifest [de949f98f03c14d798f17f843fd43beeb52b2f8b]

old_revision [b99f49b10a5135bee6185311f7f68a41c258ffab]
old_manifest [21e67aef084c054f0b4428bfe419def22d3d5e57]

patch "foo"
 from [bdca16855faf16c12b6f054813bdde0528cc356b]
   to [d686d8faedaffb518ecf7a01c1531cef2600a69b]
```

$$\downarrow$$

$$\{\text{SHA-1}\}$$

$$\downarrow$$

6714cd29a0bf86c15319199ada76851a9ab2d686

What does a development tree look like?
Meta-data (certs)

`monotone` stores meta-data along with revisions in so called certs (NOT X.509 certificates!):

- a revision ID
- a name (a cert identifier)
- a value
- a RSA key reference to the key that has signed this cert
- a RSA signature

There are some reserved certs: `author`, `branch`, `changelog`, `date`.

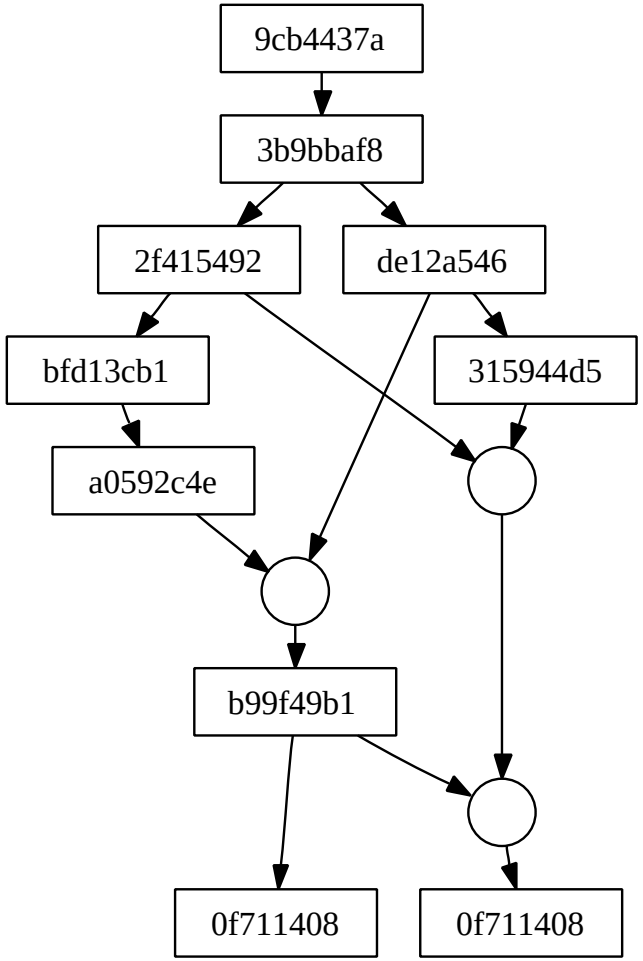
What does a development tree look like?
The concept of branches

- Everything lives in branches.
- Branches are light weight (an attribute to the revision).
- Merging between branches is called “propagating”.

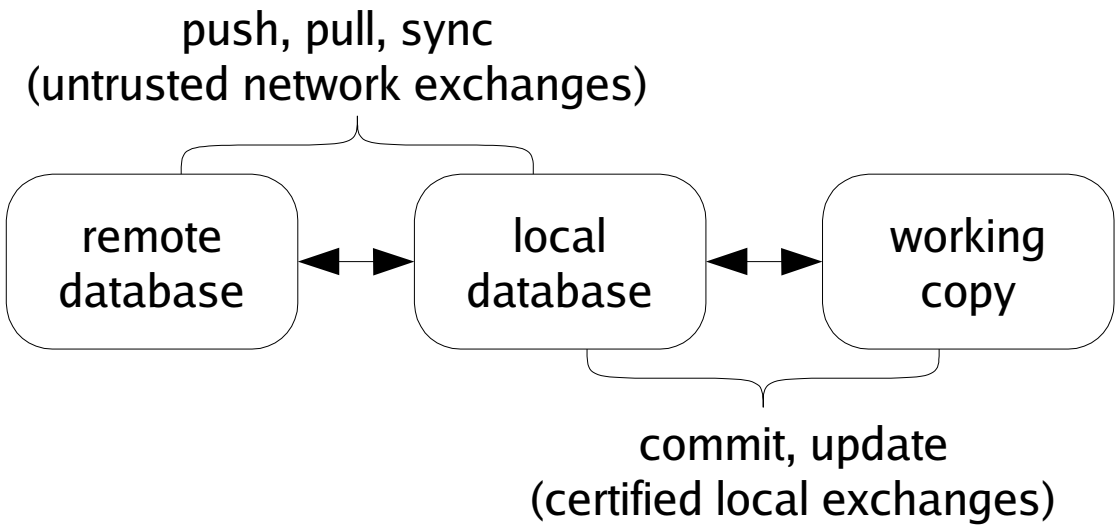
What does a development tree look like?

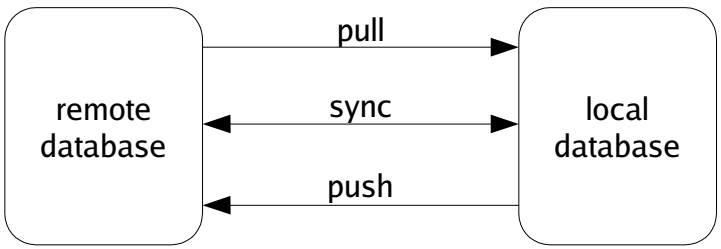
The concept of forks in the line of development

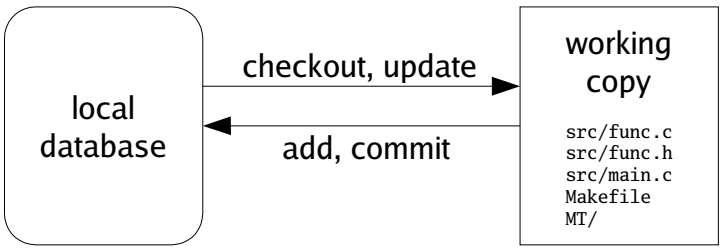
- Your local database may not always be entirely updated.
- You may lack the most recent revisions.
- When you pull new data to your database, you may find that a fork has formed.
- DON'T PANIC! This is a feature, and happens pretty commonly.
- `monotone` developers see this all the time.
- When seeing a fork, `merge`!



Workflow, storage and control
Normal workflow







Workflow, storage and control
Storage

Your work is potentially stored in three places (Who needs backups? :-)):

- in your work directory
- in your local database
- in a remote database

Your work directory has a special administrative subdirectory, `MT`. It has at least three files, `options`, `revision` and `log`.

Workflow, storage and control

How is control performed?

- Distributed means access control works differently!
- You have control over what changes get applied to your work directory.
- You do *not* have control over the changes done to anyone else's work directory.

- Control is based on your trust in the signatures.
- Control is done through programmable hooks.

- Control is done on: local commit, cert signatures, test results, network reads and network writes.

In practice

Let's see what we can do with `monotone`...

In practice
Creating a database

First, you must create your local database.

```
/home/levitte$ monotone --db=~/db.project db init

/home/levitte$ monotone --db=~/db.project genkey levitte@lp.se
monotone: generating key-pair 'levitte@lp.se'
enter passphrase for key ID [levitte@lp.se] : <enter passphrase>
confirm passphrase for key ID [levitte@lp.se]: <enter passphrase>
monotone: storing key-pair 'levitte@lp.se' in database
```

In practice
Starting a project

You start a new project by creating a work directory.

```
/home/levitte$ monotone --db=~/db.project --branch=foo.com:project \
                        setup project

/home/levitte$ ls -R project
project:
MT

project/MT:
log  options  revision
```

In practice

Starting work on someone else's project

To work on someone else's project, you pull it first!

```
/home/levitte$ monotone --db=~/db.project \
                        pull server.foo.com 'foo.com:project*'
```

Then you check out the branch you want.

```
/home/levitte$ monotone --db=~/db.project --branch=foo.com:project \
                        co project
```

In practice
Staying up to date

Staying up to date is an easy two-step operation.

```
/home/levitte/project\$ monotone pull
...
/home/levitte/project\$ monotone update
...
```

Oh, wait, did you notice something odd?

In practice
Adding files

Let's add a file to the project.

```
/home/levitte/project$ cat >> NOTES
Adding a private not just for the heck of it...
^D
/home/levitte/project$ monotone add NOTES
monotone: adding NOTES to working copy add set
```

And look, a new administrative file appeared!

```
/home/levitte/project$ ls MT
log  options  revision  work
/home/levitte/project$ cat MT/work
add_file "NOTES"
```

In practice
Committing changes

When satisfied with the changes, commit!

```
/home/levitte/project$ monotone commit -m "a commit"
enter passphrase for key ID [levitte@lp.se] : <enter passphrase>
monotone: beginning commit on branch 'foo.com:project'
monotone: committed revision 2e24d49a48adf9acf3a1b6391a080008cbef9c2
```

There's no `MT/work` any more, it's operations having been performed.

```
/home/levitte/project$ cat MT/revision
2e24d49a48adf9acf3a1b6391a080008cbef9c21
```

In practice
Taking a look at the revision data

Let's look at the meta-data that came with the committed revision.

```
/home/levitte/monotone$ monotone list certs 2e
monotone: expanded selector '2e' -> 'i:2e'
monotone: expanding selection '2e'
monotone: expanded to '2e24d49a48adf9acf3a1b6391a080008cbef9c21'
-----------------------------------------------------------------
Key   : levitte@lp.se
Sig   : ok
Name  : branch
Value : foo.com:project
-----------------------------------------------------------------
Key   : levitte@lp.se
Sig   : ok
Name  : date
Value : 2004-10-26T02:53:08
-----------------------------------------------------------------
Key   : levitte@lp.se
Sig   : ok
Name  : author
Value : levitte@lp.se
-----------------------------------------------------------------
Key   : levitte@lp.se
Sig   : ok
Name  : changelog
Value : a commit
```

In practice
Pushing your changes

If you want to push your changes to a remote server, you need to send your public key to it's administrator so he/she can give you access.

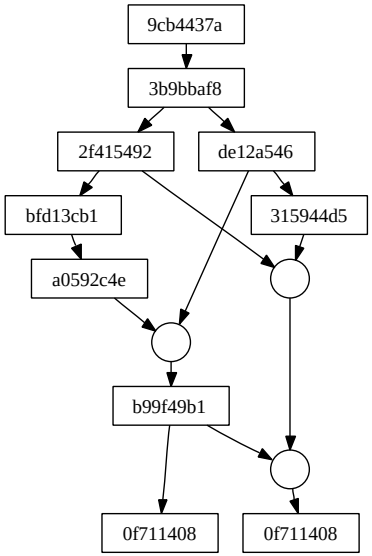
```
/home/levitte/project$ monotone pubkey levitte@lp.se > ~/levitte.pubkey
```

```
\footnotesize
/home/levitte/project$ cat ~/levitte.pubkey
[pubkey levitte@lp.se]
MIGdMA0GCSqGSIb3DQEBAQUAA4GLADCBhwKBgQC2CmCt662Ci9hff7R0YL6n02kksLlEU/+e
2V70s73pYmdFtFTjATYUVgVLV24TdXm5TQaVho4WWzGzGeYtcax4IjLBUo0uzznky4iZLei7
XfLDdFyS3+c4f1DXNx7OA3HkAuyHrxveOnqfMuQzUZoswwTue2Rhx3JUEndi2ubKoQIBEQ==
[end]
```

After you have access, all you need is to push.

```
/home/levitte/project$ monotone push
enter passphrase for key ID [levitte@lp.se] : <enter passphrase>
...
```

In practice
Dealing with a fork




```
/home/levitte/project$ EDITOR=emacs monotone merge
monotone: starting with revision 1 / 2
monotone: merging with revision 2 / 2
monotone: [source] 0f711408dfddd6afa65e9e3f5619d38d250bd09f
monotone: [source] 6714cd29a0bf86c15319199ada76851a9ab2d686
monotone: common ancestor b99f49b10a5135bee6185311f7f68a41c258ffa
b levitte+project@lp.se 2005-09-29T21:45:53 found
monotone: trying 3-way merge
monotone: help required for 3-way merge
monotone: [ancestor] foo
monotone: [    left] foo
monotone: [   right] foo
monotone: [  merged] foo
executing external 3-way merge command
enter passphrase for key ID [levitte+project@lp.se]:
monotone: [merged] 4b3cd3ee5682aa7f5865c4728ea89fd2a7dbba1a
monotone: note: your working copies have not been updated
```

In practice
Branching

Time to create a branch in the development:. First, we need to move to a starting point.

```
/home/levitte/project$monotone update -r b99
monotone: expanded selector 'b99' -> 'i:b99'
monotone: expanding selection 'b99'
monotone: expanded to 'b99f49b10a5135bee6185311f7f68a41c258ffab'
monotone: selected update target b99f49b10a5135bee6185311f7f68a41c258ffab
monotone: updating foo to bdca16855faf16c12b6f054813bdde0528cc356b
monotone: updated to base revision b99f49b10a5135bee6185311f7f68a41c258ffab
```

And then we do a reformatting change and commit it to the new branch.

```
/home/levitte/project$ monotone ci -b lp.se:testbed.project.reformat \
                                -m "Reformat"
monotone: beginning commit on branch 'lp.se:testbed.project.reformat'
enter passphrase for key ID [levitte+project@lp.se]:
monotone: committed revision 28e73a329fc2566a734da05521bf51ffdc79dd2b
```

In practice

Propagating from onte branch to another

At some point, you might want to make sure your branch us updated with the latest changes from the main line of development.

```
/home/levitte/project$ EDITOR=emacs monotone propagate \
                                              lp.se:testbed.project \
                                              lp.se:testbed.project.reformat
monotone: propagating lp.se:testbed.project -> lp.se:testbed.project.reformat
monotone: [source] 4b3cd3ee5682aa7f5865c4728ea89fd2a7dbba1a
monotone: [target] 28e73a329fc2566a734da05521bf51ffdc79dd2b
monotone: common ancestor b99f49b10a5135bee6185311f7f68a41c258ffab levitte+pr
oject@lp.se 2005-09-29T21:45:53 found
monotone: trying 3-way merge
monotone: help required for 3-way merge
monotone: [ancestor] foo
monotone: [    left] foo
monotone: [   right] foo
monotone: [  merged] foo
executing external 3-way merge command
enter passphrase for key ID [levitte+project@lp.se]:
monotone: [merged] df2f4d07675b0089d6b04864bc30cfe8a98447b4
```

A word on uniqueness and world-wide distribution

- A repository is potentially distributed world-wide.
- A repository is potentially merged together with other repositories in a single database.
- You risk name clashes!

To solve this problem, branch names, tag names and key identities need to be unique world-wide. There are conventions and proposals to do just that.

A word on uniqueness and world-wide distribution

Naming a branch

The general convention is that branches and sub-branches are separated with periods.

Example: `foo.bar.cookies`, which is a sub-branch to `foo.bar`

This isn't globally unique!

Current convention for globally unique branch names:

RFQDN`.`*branch*`[.`*subbranch*`[...]]`

An alternate proposal that separates the host name from the branches:

FQDN`:`*branch*`[.`*subbranch*`[...]]`

Examples: `net.venge.monotone`, `free.lp.se:X.ctwm`

A word on uniqueness and world-wide distribution

Naming a key identity

With `monotone`, you can't have several keys with the same identity!

Current convention: give each key an email address for an identity.

Example: `levitte@lp.se`

If you want to use several different keys for different projects, use an email address with a + directive added.

Example: `levitte+project1@lp.se`

Note: The key identity doesn't have to be a real working email address!

A word on uniqueness and world-wide distribution
Naming a tag

There is no convention for tag names!

Tools

There are a number of practical tools that interact with `monotone` in different ways. Here's a selection:

monotone-viz A `monotone` history visualiser, built with GTK+.

viewmtn a web interface to a `monotone` repository.

mtsh GTK+ wrapper for `monotone` focusing on working copy operations – add, drop, revert, rename, commit, update, diff, and browsing. Has a mechanism for per-file commit comments.

shell completion `monotone` ships with completion scripts for both bash and zsh, in the contrib/ directory of `monotone`'s source tree.

RSCM::Monotone a ruby interface to `monotone`.

monotone-notify.pl A script to watch a `monotone` repository and, for example, send emails on commits. In contrib/ directory of `monotone`'s source tree.

Where to go next

This was just a short presentation of `monotone`. There's a lot more, and if you want to know more, a good starting point is to pick up the manual (`http://www.venge.net/monotone/monotone.pdf`).

`http://www.venge.net/monotone/`
: The source of all things `monotone`.

`http://www.lua.org/`
: The language to program `monotone` hooks.