- 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.

- every participant has a complete snapshot of the repository.
- the local copy of the repository is synchronised with a remote server through an operation separate from committing a change.
- because every participant always has a complete copy of the repository, every participant can set up a server of his/her own at any time.
- because of this, there's no dependency on a single central server (unless there's only one official server, of course).
- some sort of unique revision identity. With `monotone`, the choice fell on SHA-1 hash values of the revision data and file data.
- Forget the streamlined update-then-commit model that CVS, SVN and a few more centralised SCMs. Several lines of development can exist in parallell within a branch.
- It's actually encouraged to commit your changes first and update afterwards. If a fork appeared, you can always merge.

- 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.

- monotone doesn't depend on any external database server. The repository database is managed with SQLite.
- monotone doesn't depend on any external communication server. It has it's own communication protocol (an adaptation of rsync).
- monotone doesn't need any special privileges, except for the monotone port (was 5253, will soon be 4961 through IANA assignment). It can be used as-is by any user.

- All changes to the repository are signed cryptographically.

- Everything that's placed in the repository is signed using RSA keys. Changes, files, meta-data alike.

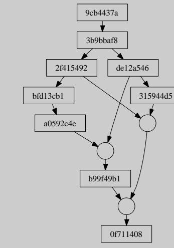
- There are two common views on change history: per-file and per-change.
- `monotone` uses the per-change view.
- All operations that change anything are atomic.
- All operations that change anything are rolled back on error.

- There's a political battle going on if revisions should be regarded as a per-file property (as it is with CVS, Digital CMS, as well as their inspiration like RCS, SCCS and the like) or a per-change property (as it is in almost all more modern SCMs).
- Since politics isn't really the purpose of this lecture, I'll just say that `monotone` regards revisions as per-change properties, and is therefore called changeset-oriented.
- An important property with `monotone` is that commits, as well as anything else that changes the repository database, are atomic and will roll back completely if an error occurs, therefore always leavin the repository database in a consistent state.

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

monotone has everything in branches.
In some SCMs, like CVS and SVN, there's the concept of a main trunk or base. monotone doesn't have any such concept. Instead, it's up to the user to define what branch is to be considered the main trunk and inform all other developers.

Likewise, CVS has a concept of vendor branches, where external source is tracked. Again, monotone doesn't have any such concept, and leaves it up to the user to define such a branch and handle external source accordingly. There are scripts that can be used to implement this easily.



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

The development history in a repository is a directed acyclic graph (DAG). This is implemented by having every revision include the identities of it's parents.

- 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).

- how revisions are formatted, how the revision IDs are created and such are very well documented in the manual.
- There are not (should not be, for the moment) any corner case that behaves in an inconsistent way with the general way `monotone` works.
- merges gives consideration to previous history, and a remerge of the same two lines of development therefore don't become conflict-loaded messes as it does with other SCMs (CVS and SVN alike, for example).

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]
```

↓
{SHA-1}
↓
6714cd29a0bf86c15319199ada76851a9ab2d686

The revision is the basis of source control with `monotone`. Every revision expresses what changes have been made since the parent revision or revisions (in case of a merge or propagate), and refers back to it's parents. Every revision is always considered unique.

A quote from another `monotone` developer: "the last revisionid can be used to prove that all previous revisions existed"

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.

monotone can store meta-data along with revisions, some of it automatically, in something called certs (NOT to be confused with X.509 certificates). Viewed another way, you could say that each cert is a statement that someone does about this revision. A cert is basically a tuple consisting of the following:

- 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

Among other things that are stored as certs, you always find the commit author, the commit date, the branches this revision belongs to and the change log.

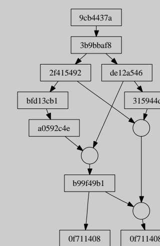
- Everything lives in branches.
- Branches are light weight (an attribute to the revision).
- Merging between branches is called "propagating".

In `monotone`, everything lives in branches. There's no main trunk, it's up to the developpers to decide which branch is to be regarded as a main trunk.

Also, branches are light-weight, and are really just an attribute attached to each revision.

In `monotone`, merging changes from one branch to another is called "propagating".

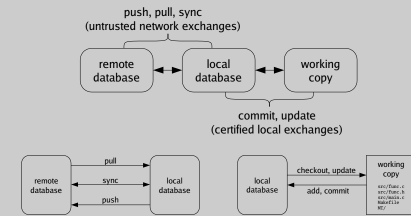
- Your local database may not always be entirely updated.
- You may lack the most recent revisions.
- When you pull new data to your database, you may find that a fork has formed.
- DON'T PANIC! This is a feature, and happens pretty commonly.
- `monotone` developers see this all the time.
- When seeing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



As a consequence of the distributed nature of `monotone`, you can never be sure that any server has all the history at every moment in time. It is therefore perfectly possible to have several changes coming from the same revision, or in other words, any revision may have more than one child. This is called a fork in a branch, or a multiheaded branch. Because of this, it may be necessary for someone to merge the heads together at some well chosen moment.

The development tree for `monotone` has these forks and merges happening all over the place.

To those who are used to the clean update-then-commit model that CVS follows, the mesh of development that will emerge with `monotone`'s model may seem scary, but really isn't. The only current problem is that conflicts have to be fixed as part of the merge, there's no possibility to have the conflicting source in your work directory and do the work in a calmer manner. THIS IS BEING WORKED ON AS WE SPEAK!



Because commits can be done off-line, there's a separation between the revision workflow (commits and updates) and the network workflow. Basically, you commit from and update your work directory from a local database, which you synchronise separately with a remote server (using push, pull or sync).

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`.

So your work is potentially stored in three places:
- in your work directory. - in your local database. - in a remote database.
Who needs backups? :-)

Your work directory has one special subdirectory that's always ignored when committing, called `MT`. It's used for administration purposes and as temporary storage for some complex operations, and you may use it for your own fiddling if needed (BUT BE CAUTIOUS!).

- 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.

Again, because of the distributed nature of monotone, access control can't really be done in a "traditional" manner as it's done in centralised SCMs. One view of a monotone repository is that it's simply a placeholder for information, and that it's up to everyone who want to pull data to decide what he/she trusts.
So basically, the decision on who and what to trust is left with every participant. It is possible to control what get's into a database on a server by limiting who has read and write access.
It should be noted that there are talks about some ACL like functionality built into monotone, but nothing real has emerged yet.
Control is performed through hooks to user-defined functions, covering the following:

- local commits (done during commit) - cert signatures (done during update) - test results (done during update) - netsync reads (done during serve when someone does a pull or sync) - netsync writes (done during serve when someone does a push or sync)

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
```

Before you do anything else with `monotone`, you must create the database. If you intend to commit anything, you also need to create a key pair.

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
```

Starting a new project involves deciding on a new branch name and creating a work directory.
When that's done, there's the work directory `project` with a subdirectory `MT`.
The files in project/MT are administrative files. `revision` contains the current revision identity. `options` contains information of all sorts, like the current database and branch for that work directory.

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
```

If you want to start working (or even just looking at someone else's project), you need to pull a copy of the database, then check out the chosen branch into a work directory.
Note that the last argument has a wildcard because I want to get all branches that start with `foo.com:project` (so I get sub-branches as well). If I only want the `foo.com:project` branch while ignoring all sub-branches (which might have slightly surprising results, btw :-)).
The second step is to check out the stuff.

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?

When you collaborate with others, you need to pull changes made by others and update your work directory.

Note that we didn't need to give `monotone pull` any information about what server to connect with or what branch pattern to pull! This is because the information was stored in the database when the first `monotone pull` was done, in special database variables that are used to store defaults.

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"
```

OK, now that we have a working directory, let's start to do some work. How about adding a file?

When the file is added, you will see that there's a new file in the `MT` directory, `work`. This is how `monotone` keeps track of changes that involves adding, renaming and deleting files, i.e. changes that can't be expressed with a diff.

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 2e24d49a48adf9acf3a1b6391a080008cbef9c21
```

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

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

When done with the change, you need to commit it.
Did you notice, btw, that we're not giving the –db option any more?
That's right, you don't need to do that, since there's information in `MT/options` saying what database you're working against.

If you look at `MT/revision`, you will see that it contains the hexadecimal revision ID shown by the commit command.

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
```

For your information, it is good to look at the information that `monotone` has stored beside the file changes.

You may note that I shortened the revision. This is part of `monotone`'s automatic expansion of some arguments, most notably revision identities. It basically works as long as the given argument leads to a unique revision.

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+c4f1DXNx70A3HkAuyHrxve0nqfMuQzUZoswwTue2Rhx3JUEndi2ubKoQIBEQ==
[end]
```

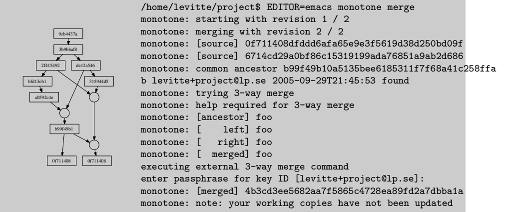
After you have access, all you need is to push.

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

I will not take up how to set up a server in this lecture, there's not enough space in this lecture for that kind of operation. However, pushing your changes to another server is easy enough.

First of all (and this is a one-time operation), you need to extract the public half of your key pair and send it to the owner of the server. The public key is just a text file which can be viewed with any text processor.

When you have received notice that you now have access, all you need to do is push



So we finally got to the point where there's a fork in the line of development. No problem, just merge!

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
```

Branching is quite easy, just commit to the new branch!

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 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.

There's absolutely noone stopping you from pulling from a lot of different sources into the same database! This means that for anyone who plans to use `monotone` for a public project or a project that just might become public one day, it's crucial to think about the uniqueness of your key identities and branch and tag names.

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`

Branches are generally named as a series of sub-branches separated with dots, so for example, `foo.bar.cookie` would be a sub-branch of `foo.bar`. There is nothing really stopping you from using a different convention, except you will confuse the hell of the rest of the world :-). And this isn't necessarely globally unique.

There's currently one name convention to solve this, which is to use a (your) reversed domain name and tuck the name of your project at the end. For example, `monotone` is created by the owner of the domain venge.net, so the main branch has been called `net.venge.monotone`.

There has been some talk about reworking this solution so it separates the host/domain part from the actual branch. I've proposed the format `{domain}:{branch}`, and am using it for my own projects, for example `free.lp.se:X.ctwm`.

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!

RSA keys are generally given an email like identity, like `foo@bar.com`. Again, there's nothing really stopping you from using a different convention, and the consequence is the same.

There is no convention for tag names!

You may notice that I still have said nothing about tag names, and that's because noone has even talked about it yet. It's just been my experience that you also need to think about how you want to name tags, unless you want to do the same stupidity I did, to name them `v{n}.{m}` for several projects that eventually ended up in the same repository!