#### ♦ Home | Documentation | Developer Document Catalog (2.2b1) »

next

# Bazaar Developer Document Catalog

## Introduction

Contributing to Bazaar

## Working on Bazaar

- Bazaar Release Cycles
- Profiling
- Tracking Bugs in Bazaar
- Bazaar Developer Guide
- Bazaar Testing Guide
- Contributing to Bazaar Documentation (wiki)

## **Releasing and Packaging**

- Releasing Bazaar
- Managing the Bazaar PPA
- Bazaar Windows EC2 Server

## Developing using bzrlib

- Bazaar Architectural Overview
- Integrating with Bazaar
- Writing plugins for Bazaar (web link)
- bzrlib API reference (web link)

## Other documents

- Bazaar Design Principles
- Plans
- Specifications
- Implementation notes
- Miscellaneous notes

♦ Home | Documentation | Developer Document Catalog (2.2b1) »

next

In the second se

# Contributing to Bazaar

## Talk to us

If you want to fix or improve something in Bazaar, we want to help you. You can ask at any time for help, on the list, on irc, or through a merge proposal on Launchpad.

In particular, the rostered Patch Pilot is an experienced developer who will help you get your changes in, through code review, advice, debugging, writing tests, or whatever it takes.

- Bazaar mailing list
- IRC in channel #bzr ON irc.ubuntu.com

## Starting

Before starting on a change it's a good idea to either file a bug, find a relevant existing bug, or send a proposal to the list. If there is a bug you should set it to "In Progress" and if you wish assign it to yourself.

You might like to start with a bug tagged easy.

### Making a branch

First, get a local copy of Bazaar:

```
$ cd $HOME
$ bzr init-repo bzr
$ cd bzr
$ bzr branch lp:bzr bzr.dev
```

Now make your own branch; we recommend you include the bug number and also a brief description:

\$ bzr branch bzr.dev 123456-status-speed

and go ahead and commit in there. Normally you should fix only one bug or closely-related cluster of bugs per branch, to make reviews and merges flow more smoothly.

For bugs that exist in older supported branches of bzr like 2.0 or 2.1, you might want to fix the bug there so it can go into a bugfix release, ie

```
$ bzr branch lp:bzr/2.1 bzr.2.1
$ bzr branch bzr.2.1 123458-2.1-status
```

You probably want this configuration in ~/.bazaar/locations.conf:

```
[/home/USER/bzr]
push_location = lp:~LAUNCHPAD_USER/bzr/
push_location:policy = appendpath
public_branch = http://bazaar.launchpad.net/~LAUNCHPAD_USER/bzr
public_branch:policy = appendpath
```

with your local and Launchpad usernames inserted.

### Writing tests

We value test coverage and generally all changes should have or update a test. There is a powerful test framework but it can be hard to find the right place to put your test. Don't hesitate to ask, or to propose a merge that does not yet have tests.

Normally for command-line code you should look in bzrlib.tests.blackbox and for library code in bzrlib.tests. For for which functions on an interface there are multiple implementations, like *Transport*, look in bzrlib.tests.per\_transport.

It's a good idea to search the tests for something related to the thing you're changing and you may find a test you can modify or adapt.

To run the tests:

\$ ./bzr selftest

Normally the tests will skip if some library dependencies are not present. On Ubuntu, you can install them with this command (you must have some repositories enabled in Software Sources):

\$ sudo apt-get build-dep bzr

To build the binary extensions:

\$ make

For more information: Testing Guide.

### Proposing a merge

Then propose a merge into bzr; for bzr 2.2 and later you can use the bzr propose-merge command. In the comment for your merge proposal please explain what you're trying to do and why. For example:

As discussed on the mailing list, this patch adds a What's New document summarising the changes since 2.0.

If you make additional changes to your branch you don't need to resubmit; they'll automatically show up in the merge proposal.

• Launchpad Code Review Help.

previous | next

In the second se

## Bazaar Release Cycles

status: Current policy, as of 2009-08. blueprint: <a href="https://blueprints.launchpad.net/bzr/+spec/6m-cycle>">https://blueprints.launchpad.net/bulta

Our users want easy access to bug fixes without other changes to the core product. They also want a Just Works experience across the full Bazaar ecosystem. To deliver the first and enable the second, we're adopting some standard process patterns: a 6 monthly release cycle and a stable branch. These changes will also have other benefits, including better availability of bug fixes in OS distributions, more freedom to remove old code, and less work for in packaging.

See also:

- Bazaar Developer Document Catalog
- Releasing Bazaar the process for actually making a release or release candidate.

### The Process

Bazaar will make a major release every six months, which will be supported at least until the time of the next major release. During this support period, we'll make incremental releases which fix bugs, but which do not change network or disk formats or command syntax, and which do not require updates to plugins.

We will also run a development series, which will become the next major release. We'll make a beta release from this every four weeks. The beta releases will be as stable as our current monthly releases and completely suitable for everyday use by users who can tolerate changes from month to month.

Having the stable series isn't a reason to cut back on QA or to make the trunk or development releases unstable, which would only make our job harder. We keep our trunk in an always-releasable state, and that should continue: any beta release could potentially be supported in the long term, but we identify particular releases that actually will be supported.

The trunk will never be frozen: changes that pass review, other quality checks and that are agreed amongst the developers can always be landed into trunk. The only restrictions will be on branches specifically targeted at a release.

#### Schedule

```
2.0.0 --- 2.0.1 -- 2.0.2 -- ...
+--2.1.0beta1 -- 2.1.0beta2 -- ... -- 2.1.0rc1 -- 2.1.0 -- 2.
\
+-- 3.0.
```

Starting from the date of a major release:

At four-week intervals we make a new beta release. There will be no separate release candidate, but if a serious problem is discovered we may do the next beta ahead of schedule or make a point release. There will be about five or six releases in that series.

In parallel with this, bugs targeted to the previous major release are merged into its branch. We will make bugfix releases from that branch as appropriate to the accumulation of changes, perhaps monthly, perhaps more often if there are serious bugs, perhaps much less often if no new changes have landed.

We will then make a release candidate for the next major release, and at this point create a release branch for it. We will iterate release candidates at approximately weekly intervals until there are no bugs blocking the final major release.

Compared to the current process this has approximately the same amount of release-related work, because the extra releases from the stable branch are "paid for" by not doing RCs for the development series.

We will synchronize our major releases with Ubuntu, so that they come out in sufficient time for some testing and margin of error before Ubuntu's upstream freeze.

#### Regularity

We value regular releases. We prefer to slip a feature or fix to a later release rather than to make a release late. We will normally only slip a release to fix a critical bug.

#### Numbering

The number for a six-month cycle is chosen at the start, with an increment to either the first field (3.0.0) or second field (3.1.0) depending on what we expect to be the user impact of the release. We expect releases that culminate in a new disk format or that require changes in how people use the tool will get a new major number. We can change (forward only) if it turns out that we land larger changes than were expected.

We will always use the 3-digit form (major.minor.micro) even when referring to the initial major release. This should help clarify where a patch is intended to land. (eg, "I propose this for 2.0.0" is clear, while "I propose this for 2.0" could mean you want to make the 2.0.0 release, or that you just want to land on the 2.0.x stable release series.)

#### Terminology

```
Major releases (2.0.0 or 2.1.0)
```

The big ones, every six months, intended to ship in distributions and to be used by stability-oriented users.

Release candidate (2.0.0rc1)

A preview of a major release, made one or a few weeks beforehand at the time the release branch is created. There should be few if any changes from the rc to the stable release. We should avoid the confusing phrasing "release candidate 2.0.0rc1 is released"; instead use "available."

Bugfix releases (2.0.1)

Based on the previous major release or bugfix; contains only bugfixes and perhaps documentation or translation corrections.

Stable series

A major release and its descendant bugfix releases.

Stable release

Either a major release or a bugfix release.

Beta release (3.0.0beta1)

Made from trunk every month, except for the month there's a major release. Stable and suitable for users who want the latest code and can live with some changes from month to month.

**Development series** 

The development releases leading up to a stable release.

#### Bug Work

Bug fixes should normally be done first against the stable branch, reviewed against that branch, and then merged forward to trunk.

It may not always be easy to do this, if fixing the bug requires large changes or the affected code is different in the stable and development branches. If the tradeoff does not seem worthwhile the bug can be fixed only in the development branch, at least in the first instance. If users later want the fix backported we can discuss it.

Developers can merge the release branch into trunk as often as they like, only asking for review if they're making nontrivial changes or feel review is needed.

#### Feature and Performance Work

Features can be landed to the development branch at any time, and they'll be released for testing within a month.

Performance bugs, although important, will generally not be landed in a stable series. Fixing performance bugs well often requires nontrivial code changes or new formats. These are not suitable for a stable series.

Performance bugs that can be fixed with a small safe patch can be considered for the stable series.

#### Plugins

Plugins that want to cooperate with this should make a series and a branch that matches each bzr stable series, and follow similar rules in making releases from their stable branch. We'd expect that plugins will make a release between the last development release of a series and the major release candidate.

Within a stable series, anything that breaks any known plugin is considered an API break and will be avoided. Before making each bugfix release, we'll test that code against important plugins.

Within a development series, the focus is on helping plugin authors keep up to date by giving clear error messages when an interface is removed. We will no longer focus on letting old plugin code work with new versions of bzrlib, which is an elusive target in Python.

This may mean that in cases where today a plugin would keep running but give warnings, it will now fail altogether with an error.

In return we expect more freedom to change and cleanup bzrlib code without needing to keep old code around, or write extra compatibility shims, or have review turnarounds related to compatibility. Some changes, such as removing module-global variables, that are hard to do now, will be possible to do safely.

Discussion of plugins here includes programs that import and use bzrlib but that aren't technically plugins. The same approach, though the technical considerations are different, should apply to other extensions such as programs that use bzr through the shell interface.

#### Data and Network Formats

Any development release should be able to interoperate with the

previous stable release, and any stable release should be able to interoperate with the previous stable release. This is a minimum and normally releases will be able to interoperate with all previous releases as at present.

Each major release will have one recommended data format which will be the default. The name of the format will indicate which release series (not specific release) it comes from: '2a' is the first supported format for the 2.0.x series, '2b' the second, etc. We don't mention the particular release that introduced it so as to avoid problems predicting precisely when it will land.

During a development series we may have a series of experimental formats. We will not leave people stranded if they test these formats, but we also won't guarantee to keep supporting them in a future release. If something inserted in one development release turns out to be bad it can just be removed in the next.

#### Hosting Services

The guarantees made above about format and network interoperation mean that hosting services such as Launchpad, Savannah, FedoraHosted, and Sourceforge could choose to run either the stable or beta versions. They might find it useful to run the beta version on their own beta server.

#### Simultaneous Installation

Some people may want to simultaneously install and use both a stable release and development release.

This can be handled in various ways either at the OS packaging or the Python level. We don't propose to directly address it in the upstream source. (For example, we will not change the bzrlib library name from one release to the next.) The issue already exists with people who may want to use for example the previous bzr release and the trunk. There is a related issue that plugins may be compatible with only some of the Bazaar versions people want to use at the same time, and again that is something that can be handled separately.

#### OS Distributions

OS distributors will be recommended to ship the bzr stable release that fits their schedule, the betas leading up to that release during their own beta period, and the bugfix releases following on from it. They might also choose to offer the beta releases as an alternative package.

#### Packaging

At present we have three upstream-maintained PPAs containing Ubuntu packages of Bazaar: ~bzr-nightly-ppa, ~bzr-beta-ppa (rcs and releases) and ~bzr (ie stable). We will keep these PPAs, and reorient beta to contain the monthly beta releases, and the stable PPA to contain stable releases, their release candidates, and bugfixes to those releases.

Some platforms with relatively less active packagers may choose to ship only the stable releases. This is probably better than having them only intermittently or slowly ship the monthly releases.

Binary installers should use a version number like '2.0.0-1' or '2.0.0beta1-1' so that the last component just reflects the packaging version, and can be incremented if a new installer is made with no upstream source changes.

#### Code Freeze vs Announcement

We will separate the code freeze for a particular release from its actual announcement, allowing a window of approximately one week for plugins to be released and binary installers to be built. On the date the announcement is published, people will be able to easily install it.

#### Weekly Metronome Mail

Every week the release manager should send a mail to the Bazaar list covering these points (as appropriate):

- Early communication about changing dependencies or defaults
- Reminder re lifecycle and where we're up to right now, in particular the dates for the next release and/or candidate.
- Summary of recent successes and pending work.
- Reminder re release objectives
- Reminder re things needing attention, e.g. bug triage, reviews, testing of certain things, etc.

## Questions

Do users actually want this?

Apparently yes, because it's often requested and often raised as a problem.

Would this confuse users?

It shouldn't, because it's a fairly standard scheme.

Won't it take more time to fix bugs in multiple places? It shouldn't, because we'll only do this when the stable bugfix seems economical. When we fix bugs today in both trunk and release branches it normally does not take much more time.

What about bzr in Ubuntu LTS, with a five-year support life? Most bugs are either fixed within six months, or not fixed at all, or not very important, or fixed as part of a large rework of the code that would be too large to backport. However, if there are fixes that are especially desired in an old release and feasible to do, we can do them without making a general commitment.

Will anyone test the beta releases?

Probably yes, our most active users will run them, but if people would really rather not test them, forcing them is not helpful.

Isn't this a step backwards to a slower, less-agile process?

No, our trunk stays releasable, and we ship every month. We're just cutting out things that hold us back (continuous rather than episodic API stability; RCs every month) and giving users what they demand.

How about calling the monthly releases "milestone" or "next" not "beta"?

Those words are less scary but they also have less clear meanings.

## Expected Benefits

If this plan works, we'll expect to see the following changes. If they don't occur, we'll think again:

- We see a distribution curve of users and bug reports across nightly, monthly and stable releases, indicating that each has value.
- API changes are easier or safer to make during beta periods, without being held back by fears of compatibility or
- The stable releases are actually stable and don't introduce regressions or break plugins.
- Many bugs are fixed in stable branches, without developers feeling this is a waste of time.
- Distributions ship the stable releases in their stable releases and the bugfix releases in their bugfix releases.
- Plugin authors follow this policy, making their own bugfix releases.
- Users like it.

After doing this for the 2.0 cycle (September 2009 through to early 2010), it seems to be going well.

## Reviewing for the Stable Branch

These are guidelines and can be interpreted case-by-case.

- All changes to the stable branch should fix a bug, even if you would not normally file a bug for the change. The bug description should if at all possible explain how to manually verify the bug in a way that will fail before and pass after the change. (These are requirements for the SRU process.)
- The change should be reasonably small and conservative.
- Remember that the patch will be read during the SRU process and so keeping the patch small is useful even beyond keeping the logical changes small. Avoid doing mechanical bulk changes on the stable branch.
- Use particular care for things that may behave differently across platforms, encodings or locales. It's harder to thoroughly test these things before a release.
- Generally speaking, just cleaning things up is not a sufficient reason to make changes to the stable branch. It has to actually fix a bug.
- Changes to the stable branch should include tests as usual.
- Don't change or remove existing APIs that might be used by plugins, even if they are underscore-prefixed. Adding APIs that are also being added to the trunk branch may make sense.
- Keeping consistency with trunk is useful, but less important than keeping the stable branch stable.
- (more items welcome)

## References

1. List thread "[rfc] six-month stable release cycles", July 2009.

• Home | Documentation | Developer Document Catalog (2.2b1) » previous | next

In the second se

# Profiling

## Using profilers

Bazaar has some built-in support for collecting and saving profiling information. In the simpliest case, the --lsprof option can be used as shown below:

bzr --lsprof ...

This will dump the profiling information to stdout before exiting. Alternatively, the --lsprof-file option can be used to specify a filename to save the profiling data into to. By default, profiling data saved to a file is a pickled Python object making it possible to reload the data and do with it what you will. For convenience though:

- if the filename ends in .txt, it will be dumped in a text format.
- if the filename either starts with callgrind.out or ends with .callgrind, it will be converted to a format loadable by the KCacheGrind visualization tool.

Note that KCacheGrind's Open Dialog has a default filter than only shows files starting with callgrind.out so the longer filename is usually preferable. Here is an example of how to use the --lsprof-file option in combination with KCacheGrind to visualize what the status command is doing:

```
bzr --lsprof-file callgrind.out.st001 status
kcachegrind callgrind.out.st001 &
```

**Note:** bzr also has a --profile option that uses the hotshot profiler instead of the lsprof profiler. The hotshot profiler can be useful though the lsprof one is generally recommended. See http://docs.python.org/lib/node795.html.

Note that to use --1sprof you must install the lsprof module, which you can get with:

svn co http://codespeak.net/svn/user/arigo/hack/misc/lsprof

## Profiling locks

Bazaar can log when locks are taken or released, which can help in identifying unnecessary lock traffic. This is activated by the -Dlock global option.

This writes messages into ~/.bzr.log. At present this only logs actions relating to the on-disk lockdir. It doesn't describe actions on in-memory lock counters, or OS locks (which are used for dirstate.)

## Profiling HPSS Requests

When trying to improve network performance, it is often useful to know what requests are being made, and how long they are taking. The -Dhpss global option will enable logging smart server requests, including the time spent in each request.

Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

In the second se

# **Tracking Bugs in Bazaar**

This document describes the bug-tracking processes for developing Bazaar itself. Bugs in Bazaar are recorded in Launchpad.

See also:

- Bazaar Developer Documents.
- The Bazaar Development Cycle.
- The Bazaar User Guide for information on integrating Bazaar with other bug trackers.
# Links

- bzr bugs home page.
- Critical bugs.
- Open bugs by importance.
- Open bugs most recently changed first.
- Most commonly duplicated bugs.

### Generalities

Anyone involved with Bazaar is welcome to contribute to managing our bug reports. **Edit boldly**: try to help users out, assess importance or improve the bug description or status. Other people will see the bugs: it's better to have 20 of them processed and later change the status of a couple than to leave them lie.

When you file a bug as a Bazaar developer or active user, if you feel confident in doing so, make an assessment of status and importance at the time you file it, rather than leaving it for someone else. It's more efficient to change the importance if someone else feel's it's higher or lower, than to have someone else edit all bugs.

It's more useful to actually ship bug fixes than to garden the bug database. It's more useful to take one bug through to a shipped fix than to partially investigate ten bugs. You don't get credit for a bug until the fix is shipped in a release. Users like getting a response to their report, but they generally care more about getting bugs fixed.

The aim of investigating bugs before starting concentrated work on them is therefore only to:

- determine if they are critical or high priority (and should displace existing work)
- garden sufficiently to keep the database usable: meaningful summaries, and duplicates removed

It's OK to fix some bugs that just annoy you, even if they're not rationally high.

You can use --fixes lp:12345678 when committing to associate the commit with a particular bug.

If there are multiple bugs with related fixes, putting "[master]" in the title of one of them helps find it

It's often fastest to find bugs just using the regular Google search engine, rather than Launchpad's search.

Martin Pitt says:

One of the things you should not do often is to start asking questions/for more debug info and then forget about the bug. It's just a waste of the reporter's and your time, and will create frustration on the reporter side.

## Priorities

The suggested priorities for bug work are:

- 1. Fix critical bugs.
- 2. Get existing fixes through review and landed.
- 3. Fix bugs that are already in progress.
- 4. Look at bugs already assigned to you, and either start them, or change your mind and unassign them.
- 5. Take new bugs from the top of the stack.
- 6. Triage new bugs.

It's not strict and of course there is personal discretion but our work should be biased to the top of this hierarchy.

### **Clear Bugs**

Bugs should have clear edges, so that you can make a clear statement about whether a bug is fixed or not. (Sometimes reality is complicated, but aim for each bug to be clear.)

Bugs on documentation, performance, or UI are fine as long as they're clear bugs.

Examples of good bugs:

- "ValueError in frob\_foo when committing changed symlink" although there may be many possible things that could cause a ValueError there, you should at least know when you've fixed the problem described in this bug.
- "Unclear message about incompatible repositories" even though the user may not agree the new message is sufficiently clear, at least you know when you've tried to fix it.

Examples of bad bugs:

• "Commit is too slow" - how fast is fast enough to close it? "Commit reads the working tree twice" is clearer.

### **Bug Status**

New

The bug has just been filed and hasn't been examined by a developer yet.

Incomplete

The bug requires more information from the reporter to make progress.

Only set this state if it's impossible or uneconomical to make progress on the bug without that information. The bug will expire if it remains in this state for two months.

Confirmed

The bug report has been seen by a developer and we agree it's a bug. You don't have to reproduce the bug to mark it Confirmed. (Generally it's not a good idea for a developer to spend time reproducing the bug until they're going to work on it.)

Triaged

We don't use this status. If it is set, it means the same as Confirmed.

In Progress

Someone has started working on this. We can deliver the value of the work already done by finishing and shipping the fix.

The bug keeps this state from the time someone does non-trivial analysis, until the fix is merged to a release or trunk branch (when it is Fix Released), or until they give up on it (back to New or Confirmed) or decide it is Invalid or Incomplete.

Won't Fix

The behaviour complained about is intentional and we won't fix it. Needless to say, be thoughtful before using this status, and

consider if the user experience can be improved in some other way.

Invalid

The reporter was confused, and this is not actually a bug. Again, be sensitive in explaining this to the user.

**Fix Committed** 

Don't use this. If set on old bug, it probably means In Progress, with the fix waiting for review. See Launchpad bug 163694.

Fix Released

The fix for this bug is now in the bzr branch that this task is for. The branch for the default task on a bug is bzr.dev.

We use this value even though the fix may not have been been included in a release yet because all the developer activity around it is complete and we want to both avoid bug spam when releases happen, and keep the list of bugs that developers see when they look at the bug tracker trimmed to those that require action.

When setting a bug task to fix released, the bug target milestone should be set to the release the fix will be included in (or was included in, if you are updating an old bug). Don't spend too much time updating this if you don't immediately know: its not critical that it be set.

### **Bug Importance**

Critical

This is a serious bug that could cause data loss, stop bzr being usable in an important case, or represents a regression in something previously working. We should fix critical bugs before doing other work, or seriously consider whether the bug is really critical or whether the other change is more urgent.

High

This is a bug that can seriously interfere with people's use of Bazaar. We should seriously consider fixing these bugs before working on new features.

Medium

A regular bug. We'd like to fix them, but there may be a long delay.

Low

Something suboptimal that may affect an unimportant case or have a fairly easy workaround.

Wishlist

These will basically never get done.

Bugs rated Medium or lower are unlikely to get fixed unless they either pique the interest of a developer or are escalated due eg to many users being affected.

Not every existing bug is correctly rated according to this scale, and we don't always follow this process, but we'd like to do it more. But remember, fixing bugs is more helpful than gardening them.

# Assignment

Assigning a bug to yourself, or someone else, indicates a real intention to work on that bug soon.

### **Targetting Bugs**

It's possible to target a bug to a milestone, eg <<u>https://bugs.edge.launchpad.net/bzr/+milestone/1.16</u>>. We use this to help the release manager know what **must** be merged to make the release.

Therefore, we don't target bugs that we'd like to have fixed or that could be fixed in a particular release, we only target bugs that must be fixed and that will cause us to slip the release if they're not fixed. At any time, very few if any of the bugs targeted to a release should be still open. By definition, these bugs should normally be Critical priority.

### **Backports**

Sometimes we'll want to make a special point-release update (eg 1.15.1) off an already-released branch including a fix for a particular bug. To represent this, create a new bug task (ie link in the status table on the bug page) by clicking the poorly-named "Target to Release" link. Target it to the appropriate series (ie 1.15). If the bug should also prevent any point releases of that series then you should also target the new task to the appropriate milestone within that release. (See Targeting Bugs above)

This bug task then has a separate status and importance to indicate the separate work to get it into that release.

### The News File

Most bugs that are fixed should be mentioned in a NEWS file entry, including the bug number. (Exceptions might be bugs that are not at all user visible.)

### Tags

Here are some bug tags we use. In Malone tags are currently of limited use, so don't feel obliged to tag bugs unless you're finding it useful.

authentication

authenticating to servers

backport

candidate for backporting to an update of the previous release

dirstate

WorkingTree4

easy

should be possible to finish in an hour or two

hpss

bugs about the High-Performance Smart Server, i.e. bzr+ssh://, etc.

hpssvfs

bugs for causes of VFS methods of the smart server

launchpad

bugs about interactions with launchpad (typically this means bzrlib.plugins.launchpad).

locale

problems using locales other than English

memory

problems where we use too much memory for some reason

newformat

fixing this would need a new disk format

performance

bugs about performance problems.

test

needs changes to the test framework

transport

virtual filesystem for http, sftp, etc

trivial

should be very easy to fix (10-20 minutes) and easily landed: typically just spelling errors and the like

ui

bugs relating to the bzr user interface, e.g. confusing error messages.

win32

bugs that mainly affects Windows. Also there is cygwin and win98 tags for marking specific bugs.

You can see the full list of tags in use at <<u>https://bugs.edge.launchpad.net/bzr/+bugs</u>>. As of September 2008 the list is on the right.

 $\clubsuit$  Home | Documentation | Developer Document Catalog (2.2b1) » previous | next

In the second se

# Bazaar Developer Guide

This document describes the Bazaar internals and the development process. It's meant for people interested in developing Bazaar, and some parts will also be useful to people developing Bazaar plugins.

If you have any questions or something seems to be incorrect, unclear or missing, please talk to us in irc://irc.freenode.net/#bzr, or write to the Bazaar mailing list. To propose a correction or addition to this document, send a merge request or new text to the mailing list.

The latest developer documentation can be found online at http://doc.bazaar-vcs.org/developers/.

## **Getting Started**

### Exploring the Bazaar Platform

Before making changes, it's a good idea to explore the work already done by others. Perhaps the new feature or improvement you're looking for is available in another plug-in already? If you find a bug, perhaps someone else has already fixed it?

To answer these questions and more, take a moment to explore the overall Bazaar Platform. Here are some links to browse:

- The Plugins page on the Wiki http://bazaar-vcs.org/BzrPlugins
- The Bazaar product family on Launchpad https://launchpad.net/bazaar
- Bug Tracker for the core product https://bugs.launchpad.net/bzr/
- Blueprint Tracker for the core product https://blueprints.launchpad.net/bzr/

If nothing else, perhaps you'll find inspiration in how other developers have solved their challenges.

### Finding Something To Do

Ad-hoc performance work can also be done. One useful tool is the 'evil' debug flag. For instance running bzr -Devil commit -m "test" will log a backtrace to the bzr log file for every method call which triggers a slow or non-scalable part of the bzr library. So checking that a given command with -Devil has no backtraces logged to the log file is a good way to find problem function calls that might be nested deep in the code base.

### Planning and Discussing Changes

There is a very active community around Bazaar. Mostly we meet on IRC (#bzr on irc.freenode.net) and on the mailing list. To join the Bazaar community, see http://bazaar-vcs.org/BzrSupport.

If you are planning to make a change, it's a very good idea to mention it on the IRC channel and/or on the mailing list. There are many advantages to involving the community before you spend much time on a change. These include:

- you get to build on the wisdom of others, saving time
- if others can direct you to similar code, it minimises the work to be done
- it assists everyone in coordinating direction, priorities and effort.

In summary, maximising the input from others typically minimises the total effort required to get your changes merged. The community is friendly, helpful and always keen to welcome newcomers.

#### Bazaar Development in a Nutshell

One of the fun things about working on a version control system like Bazaar is that the users have a high level of proficiency in contributing back into the tool. Consider the following very brief introduction to contributing back to Bazaar. More detailed instructions are in the following sections.

#### Making the change

First, get a local copy of the development mainline (See Why make a local copy of bzr.dev?.)

```
$ bzr init-repo ~/bzr
$ cd ~/bzr
```

\$ bzr branch http://bazaar-vcs.org/bzr/bzr.dev/ bzr.dev

Now make your own branch:

\$ bzr branch bzr.dev 123456-my-bugfix

This will give you a branch called "123456-my-bugfix" that you can work on and commit in. Here, you can study the code, make a fix or a new feature. Feel free to commit early and often (after all, it's your branch!).

Documentation improvements are an easy place to get started giving back to the Bazaar project. The documentation is in the *doc/* subdirectory of the Bazaar source tree.

When you are done, make sure that you commit your last set of changes as well! Once you are happy with your changes, ask for them to be merged, as described below.

#### Making a Merge Proposal

The Bazaar developers use Launchpad to further enable a truly distributed style of development. Anyone can propose a branch for merging into the Bazaar trunk. To start this process, you need to push your branch to Launchpad. To do this, you will need a Launchpad account and user name, e.g. *your\_lp\_username*. You can push your branch to Launchpad directly from Bazaar:

\$ bzr push lp:~your\_lp\_username/bzr/meaningful\_name\_here

After you have pushed your branch, you will need to propose it for merging to the Bazaar trunk. Go to <https://launchpad.net/your\_lp\_username/bzr/meaningful\_name\_here> and choose "Propose for merging into another branch". Select "~bzr/bzr/trunk" to hand your changes off to the Bazaar developers for review and merging. Using a meaningful name for your branch will help you and the reviewer(s) better track the submission. Use a very succint description of your submission and prefix it with bug number if (lp:~mbp/bzr/484558-merge-directory needed for example). suffix with the Alternatively, you can bug number (lp:~jameinel/bzr/export-file-511987).

Why make a local copy of bzr.dev?

Making a local mirror of bzr.dev is not strictly necessary, but it means

- You can use that copy of bzr.dev as your main bzr executable, and keep it up-to-date using bzr pull.
- Certain operations are faster, and can be done when offline. For example:
  - bzr bundle
  - bzr diff -r ancestor:...
  - ∘ bzr merge
- When it's time to create your next branch, it's more convenient. When you have further contributions to make, you should do them in their own branch:

```
$ cd ~/bzr
$ bzr branch bzr.dev additional_fixes
$ cd additional_fixes # hack, hack, hack
```

#### Understanding the Development Process

The development team follows many practices including:

- a public roadmap and planning process in which anyone can participate
- time based milestones everyone can work towards and plan

around

- extensive code review and feedback to contributors
- complete and rigorous test coverage on any code contributed
- automated validation that all tests still pass before code is merged into the main code branch.

The key tools we use to enable these practices are:

- Launchpad https://launchpad.net/
- Bazaar http://bazaar-vcs.org/
- Bundle Buggy http://bundlebuggy.aaronbentley.com/
- Patch Queue Manager https://launchpad.net/pqm/

For further information, see http://bazaar-vcs.org/BzrDevelopment.

# Preparing a Sandbox for Making Changes to Bazaar

Bazaar supports many ways of organising your work. See http://bazaar-vcs.org/SharedRepositoryLayouts for a summary of the popular alternatives.

Of course, the best choice for you will depend on numerous factors: the number of changes you may be making, the complexity of the changes, etc. As a starting suggestion though:

• create a local copy of the main development branch (bzr.dev) by using this command:

```
bzr branch http://bazaar-vcs.org/bzr/bzr.dev/ bzr.dev
```

- keep your copy of bzr.dev pristine (by not developing in it) and keep it up to date (by using bzr pull)
- create a new branch off your local bzr.dev copy for each issue (bug or feature) you are working on.

This approach makes it easy to go back and make any required changes after a code review. Resubmitting the change is then simple with no risk of accidentally including edits related to other issues you may be working on. After the changes for an issue are accepted and merged, the associated branch can be deleted or archived as you wish.

### Navigating the Code Base

Some of the key files in this directory are:

bzr

The command you run to start Bazaar itself. This script is pretty short and just does some checks then jumps into bzrlib.

README

This file covers a brief introduction to Bazaar and lists some of its key features.

NEWS

Summary of changes in each Bazaar release that can affect users or plugin developers.

setup.py

Installs Bazaar system-wide or to your home directory. To perform development work on Bazaar it is not required to run this file - you can simply run the bzr command from the top level directory of your development copy. Note: That if you run setup.py this will create a 'build' directory in your development branch. There's nothing wrong with this but don't be confused by it. The build process puts a copy of the main code base into this build directory, along with some other files. You don't need to go in here for anything discussed in this guide.

bzrlib

Possibly the most exciting folder of all, bzrlib holds the main code base. This is where you will go to edit python files and contribute

to Bazaar.

doc

Holds documentation on a whole range of things on Bazaar from the origination of ideas within the project to information on Bazaar features and use cases. Within this directory there is a subdirectory for each translation into a human language. All the documentation is in the ReStructuredText markup language.

doc/developers

Documentation specifically targeted at Bazaar and plugin developers. (Including this document.)

Automatically-generated API reference information is available at <a href="http://starship.python.net/crew/mwh/bzrlibapi/">http://starship.python.net/crew/mwh/bzrlibapi/</a>>.

See also the Bazaar Architectural Overview.

### The Code Review Process

All code changes coming in to Bazaar are reviewed by someone else. Normally changes by core contributors are reviewed by one other core developer, and changes from other people are reviewed by two core developers. Use intelligent discretion if the patch is trivial.

Good reviews do take time. They also regularly require a solid understanding of the overall code base. In practice, this means a small number of people often have a large review burden - with knowledge comes responsibility. No one likes their merge requests sitting in a queue going nowhere, so reviewing sooner rather than later is strongly encouraged.

#### Review cover letters

Please put a "cover letter" on your merge request explaining:

- the reason **why** you're making this change
- **how** this change achieves this purpose
- anything else you may have fixed in passing
- anything significant that you thought of doing, such as a more extensive fix or a different approach, but didn't or couldn't do now

A good cover letter makes reviewers' lives easier because they can decide from the letter whether they agree with the purpose and approach, and then assess whether the patch actually does what the cover letter says. Explaining any "drive-by fixes" or roads not taken may also avoid queries from the reviewer. All in all this should give faster and better reviews. Sometimes writing the cover letter helps the submitter realize something else they need to do. The size of the cover letter should be proportional to the size and complexity of the patch.

### Reviewing proposed changes

Anyone is welcome to review code, and reply to the thread with their opinion or comments.

The simplest way to review a proposed change is to just read the patch on the list or in Bundle Buggy. For more complex changes it may be useful to make a new working tree or branch from trunk, and merge the proposed change into it, so you can experiment with the code or look at a wider context.

There are three main requirements for code to get in:

- Doesn't reduce test coverage: if it adds new methods or commands, there should be tests for them. There is a good test framework and plenty of examples to crib from, but if you are having trouble working out how to test something feel free to post a draft patch and ask for help.
- Doesn't reduce design clarity, such as by entangling objects we're trying to separate. This is mostly something the more experienced reviewers need to help check.
- Improves bugs, features, speed, or code simplicity.

Code that goes in should not degrade any of these aspects. Patches are welcome that only cleanup the code without changing the external behaviour. The core developers take care to keep the code quality high and understandable while recognising that perfect is sometimes the enemy of good.

It is easy for reviews to make people notice other things which should be fixed but those things should not hold up the original fix being accepted. New things can easily be recorded in the Bug Tracker instead.

It's normally much easier to review several smaller patches than one large one. You might want to use <u>bzr-loom</u> to maintain threads of related work, or submit a preparatory patch that will make your "real" change easier.

### Checklist for reviewers

- Do you understand what the code's doing and why?
- Will it perform reasonably for large inputs, both in memory size and run time? Are there some scenarios where performance should be measured?
- Is it tested, and are the tests at the right level? Are there both blackbox (command-line level) and API-oriented tests?
- If this change will be visible to end users or API users, is it appropriately documented in NEWS?
- Does it meet the coding standards below?
- If it changes the user-visible behaviour, does it update the help strings and user documentation?
- If it adds a new major concept or standard practice, does it update the developer documentation?
- (your ideas here...)

### **Reviews on Launchpad**

From May 2009 on, we prefer people to propose code reviews through Launchpad.

- <https://launchpad.net/+tour/code-review>
- <https://help.launchpad.net/Code/Review>

Anyone can propose or comment on a merge proposal just by creating a Launchpad account.

There are two ways to create a new merge proposal: through the web interface or by email.

Proposing a merge through the web

To create the proposal through the web, first push your branch to Launchpad. For example, a branch dealing with documentation belonging to the Launchpad User mbp could be pushed as

bzr push lp:~mbp/bzr/doc

Then go to the branch's web page, which in this case would be <<u>https://code.launchpad.net/~mbp/bzr/doc</u>>. You can simplify this step by just running

bzr lp-open

You can then click "Propose for merging into another branch", and enter your cover letter (see above) into the web form. Typically you'll want to merge into <u>bzr/bzr/trunk</u> which will be the default; you might also want to nominate merging into a release branch for a bug fix. There is the option to specify a specific reviewer or type of review, and you shouldn't normally change those.

Submitting the form takes you to the new page about the merge proposal containing the diff of the changes, comments by interested people, and controls to comment or vote on the change.

#### Proposing a merge by mail

To propose a merge by mail, send a bundle to merge@code.launchpad.net.

You can generate a merge request like this:

bzr send -o bug-1234.diff

bzr send can also send mail directly if you prefer; see the help.

#### **Reviewing changes**

From <https://code.launchpad.net/bzr/+activereviews> you can see all currently active reviews, and choose one to comment on. This page also shows proposals that are now approved and should be merged by someone with PQM access.

#### Reviews through Bundle Buggy

The Bundle Buggy tool used up to May 2009 is still available as a review mechanism.

#### Sending patches for review

If you'd like to propose a change, please post to the bazaar@lists.canonical.com list with a bundle, patch, or link to a branch. Put [PATCH] or [MERGE] in the subject so Bundle Buggy can pick it out, and explain the change in the email message text. Remember to update the NEWS file as part of your change if it makes any changes visible to users or plugin developers. Please include a diff against mainline if you're giving a link to a branch.

You can generate a merge request like this:

bzr send -o bug-1234.patch

A .patch extension is recommended instead of .bundle as many mail clients will send the latter as a binary file.

bzr send can also send mail directly if you prefer; see the help.

Please do **NOT** put [PATCH] or [MERGE] in the subject line if you don't want it to be merged. If you want comments from developers rather than to be merged, you can put [RFC] in the subject line.

If this change addresses a bug, please put the bug number in the subject line too, in the form [#1] so that Bundle Buggy can recognize it.

If the change is intended for a particular release mark that in the subject too, e.g. [1.6]. Anyone can "vote" on the mailing list by expressing an opinion. Core developers can also vote using Bundle Buggy. Here are the voting codes and their explanations.

approve:	Reviewer wants this submission merged.
tweak:	Reviewer wants this submission merged with small changes. (No re-review required.)
abstain:	Reviewer does not intend to vote on this patch.
resubmit:	Please make changes and resubmit for review.
reject:	Reviewer doesn't want this kind of change merged.
comment:	Not really a vote. Reviewer just wants to comment, for now.

If a change gets two approvals from core reviewers, and no rejections, then it's OK to come in. Any of the core developers can bring it into the bzr.dev trunk and backport it to maintenance branches if required. The Release Manager will merge the change into the branch for a pending release, if any. As a guideline, core developers usually merge their own changes and volunteer to merge other contributions if they were the second reviewer to agree to a change.

To track the progress of proposed changes, use Bundle Buggy. See http://bundlebuggy.aaronbentley.com/help for a link to all the outstanding merge requests together with an explanation of the columns. Bundle Buggy will also mail you a link to track just your

change.

# Coding Style Guidelines

### hasattr and getattr

hasattr should not be used because it swallows exceptions including KeyboardInterrupt. Instead, say something like

```
if getattr(thing, 'name', None) is None
```

Code layout

Please write PEP-8 compliant code.

One often-missed requirement is that the first line of docstrings should be a self-contained one-sentence summary.

We use 4 space indents for blocks, and never use tab characters. (In vim, set expandtab.)

Trailing white space should be avoided, but is allowed. You should however not make lots of unrelated white space changes.

Unix style newlines (LF) are used.

Each file must have a newline at the end of it.

Lines should be no more than 79 characters if at all possible. Lines that continue a long statement may be indented in either of two ways:

within the parenthesis or other character that opens the block, e.g.:

or indented by four spaces:

The first is considered clearer by some people; however it can be a bit harder to maintain (e.g. when the method name changes), and it does not work well if the relevant parenthesis is already far to the right. Avoid this:

but rather

```
self.legbone.kneebone.shinbone.toebone.shake_it(one,
     two,
     three)
```

or

For long lists, we like to add a trailing comma and put the closing character on the following line. This makes it easier to add new items in future:

```
from bzrlib.goo import (
    jam,
    jelly,
    marmalade,
    )
```

There should be spaces between function parameters, but not

between the keyword name and the value:

```
call(1, 3, cheese=quark)
```

In emacs:

```
;(defface my-invalid-face
   '((t (:background "Red" :underline t)))
   "Face used to highlight invalid constructs or other uglyties
   )
(defun my-python-mode-hook ()
 ;; setup preferred indentation style.
 (setq fill-column 79)
 (setq indent-tabs-mode nil); no tabs, never, I will not repea
   (font-lock-add-keywords 'python-mode
                             '(("^\\s *\t" . 'my-invalid-face) ; L
("[ \t]+$" . 'my-invalid-face) ;
                                ("^[ \t]+$" . 'my-invalid-face));
                              )
 )
(add-hook 'python-mode-hook 'my-python-mode-hook)
- 4
```

The lines beginning with ';' are comments. They can be activated if one want to have a strong notice of some tab/space usage violations.

### Module Imports

- Imports should be done at the top-level of the file, unless there is a strong reason to have them lazily loaded when a particular function runs. Import statements have a cost, so try to make sure they don't run inside hot functions.
- Module names should always be given fully-qualified, i.e. bzrlib.hashcache not just hashcache.

### Naming

Functions, methods or members that are relatively private are given a leading underscore prefix. Names without a leading underscore are public not just across modules but to programmers using bzrlib as an API.

We prefer class names to be concatenated capital words (TestCase) and variables, methods and functions to be lowercase words joined by underscores (revision\_id, get\_revision).

For the purposes of naming some names are treated as single compound words: "filename", "revno".

Consider naming classes as nouns and functions/methods as verbs.

Try to avoid using abbreviations in names, because there can be inconsistency if other people use the full name.

### Standard Names

revision\_id NOt rev\_id Of revid

Functions that transform one thing to another should be named  $x_{to_y}$  (not  $x_{2y}$  as occurs in some old code.)

#### Destructors

Python destructors (<u>del</u>) work differently to those of other languages. In particular, bear in mind that destructors may be called immediately when the object apparently becomes unreferenced, or at some later time, or possibly never at all. Therefore we have restrictions on what can be done inside them.

0. If you think you need to use a <u>\_\_\_\_de1\_\_</u> method ask another developer for alternatives. If you do need to use one, explain why in a comment.

- 1. Never rely on a <u>\_\_\_\_\_\_</u> method running. If there is code that must run, do it from a finally block instead.
- 2. Never import from inside a \_\_\_\_\_del\_\_ method, or you may crash the interpreter!!
- 3. In some places we raise a warning from the destructor if the object has not been cleaned up or closed. This is considered OK: the warning may not catch every case but it's still useful sometimes.

### Cleanup methods

Often when something has failed later code, including cleanups invoked from finally blocks, will fail too. These secondary failures are generally uninteresting compared to the original exception. So use the only\_raises decorator (from bzrlib.decorators) for methods that are typically called in finally blocks, such as unlock methods. For example, @only\_raises(LockNotHeld, LockBroken). All errors that are unlikely to be a knock-on failure from a previous failure should be allowed.

#### Factories

In some places we have variables which point to callables that construct new instances. That is to say, they can be used a lot like class objects, but they shouldn't be *named* like classes:

> I think that things named FooBar should create instances of F > called. Its plain confusing for them to do otherwise. When we > something that is going to be used as a class - that is, chec > isinstance or other such idioms, them I would call it foo\_cla > it is clear that a callable is not sufficient. If it is only > factory, then yes, foo\_factory is what I would use.

Registries

Several places in Bazaar use (or will use) a registry, which is a mapping from names to objects or classes. The registry allows for loading in registered code only when it's needed, and keeping associated information such as a help string or description.

#### InterObject and multiple dispatch

The Interobject provides for two-way multiple dispatch: matching up for example a source and destination repository to find the right way to transfer data between them.

There is a subclass InterObject classes for each type of object that is dispatched this way, e.g. InterRepository. Calling .get() on this class will return an InterObject instance providing the best match for those parameters, and this instance then has methods for operations between the objects.

```
inter = InterRepository.get(source_repo, target_repo)
inter.fetch(revision_id)
```

InterRepository also acts as a registry-like object for its subclasses, and they can be added through .register\_optimizer. The right one to run is selected by asking each class, in reverse order of registration, whether it .is\_compatible with the relevant objects.

### Lazy Imports

To make startup time faster, we use the bzrlib.lazy\_import module to delay importing modules until they are actually used. lazy\_import uses the same syntax as regular python imports. So to import a few modules in a lazy fashion do:

```
from bzrlib.lazy_import import lazy_import
lazy_import(globals(), """
import os
```
```
import subprocess
import sys
import time
from bzrlib import (
    errors,
    transport,
    revision as _mod_revision,
    )
import bzrlib.transport
import bzrlib.xml5
""")
```

At this point, all of these exist as a ImportReplacer object, ready to be imported once a member is accessed. Also, when importing a module into the local namespace, which is likely to clash with variable names, it is recommended to prefix it as \_mod\_<module>. This makes it clearer that the variable is a module, and these object should be hidden anyway, since they shouldn't be imported into other namespaces.

While it is possible for lazy\_import() to import members of a module when using the from module import member syntax, it is recommended to only use that syntax to load sub modules from module import submodule. This is because variables and classes can frequently be used without needing a sub-member for example:

```
lazy_import(globals(), """
from module import MyClass
""")
def test(x):
    return isinstance(x, MyClass)
```

This will incorrectly fail, because MyClass is a ImportReplacer Object, rather than the real class.

It also is incorrect to assign ImportReplacer objects to other variables. Because the replacer only knows about the original name,

it is unable to replace other variables. The ImportReplacer class will raise an IllegalUseOfScopeReplacer exception if it can figure out that this happened. But it requires accessing a member more than once from the new variable, so some bugs are not detected right away.

## The Null revision

The null revision is the ancestor of all revisions. Its revno is 0, its revision-id is null:, and its tree is the empty tree. When referring to the null revision, please use <u>bzrlib.revision.NULL\_REVISION</u>. Old code sometimes uses **None** for the null revision, but this practice is being phased out.

## Object string representations

Python prints objects using their <u>repr</u> method when they are written to logs, exception tracebacks, or the debugger. We want objects to have useful representations to help in determining what went wrong.

If you add a new class you should generally add a <u>\_\_repr\_\_</u> method unless there is an adequate method in a parent class. There should be a test for the repr.

Representations should typically look like Python constructor syntax, but they don't need to include every value in the object and they don't need to be able to actually execute. They're to be read by humans, not machines. Don't hardcode the classname in the format, so that we get the correct value if the method is inherited by a subclass. If you're printing attributes of the object, including strings, you should normally use %r syntax (to call their repr in turn).

Try to avoid the representation becoming more than one or two lines long. (But balance this against including useful information, and

simplicity of implementation.)

Because repr methods are often called when something has already gone wrong, they should be written somewhat more defensively than most code. The object may be half-initialized or in some other way in an illegal state. The repr method shouldn't raise an exception, or it may hide the (probably more useful) underlying exception.

Example:

## Exception handling

A bare except statement will catch all exceptions, including ones that really should terminate the program such as MemoryError and KeyboardInterrupt. They should rarely be used unless the exception is later re-raised. Even then, think about whether catching just Exception (which excludes system errors in Python2.5 and later) would be better.

### Test coverage

All code should be exercised by the test suite. See the Bazaar Testing Guide for detailed information about writing tests.

## **Core Topics**

## Evolving Interfaces

We don't change APIs in stable branches: any supported symbol in a stable release of bzr must not be altered in any way that would result in breaking existing code that uses it. That means that method names, parameter ordering, parameter names, variable and attribute names etc must not be changed without leaving a 'deprecated forwarder' behind. This even applies to modules and classes.

If you wish to change the behaviour of a supported API in an incompatible way, you need to change its name as well. For instance, if I add an optional keyword parameter to branch.commit - that's fine. On the other hand, if I add a keyword parameter to branch.commit which is a *required* transaction object, I should rename the API - i.e. to 'branch.commit\_transaction'.

(Actually, that may break code that provides a new implementation of commit and doesn't expect to receive the parameter.)

When renaming such supported API's, be sure to leave a deprecated\_method (or \_function or ...) behind which forwards to the new API. See the bzrlib.symbol\_versioning module for decorators that take care of the details for you - such as updating the docstring, and issuing a warning when the old API is used.

For unsupported API's, it does not hurt to follow this discipline, but it's not required. Minimally though, please try to rename things so that callers will at least get an AttributeError rather than weird results.

#### Deprecation decorators

bzrlib.symbol\_versioning provides decorators that can be attached to methods, functions, and other interfaces to indicate that they should no longer be used. For example:

```
@deprecated_method(deprecated_in((0, 1, 4)))
def foo(self):
    return self._new_foo()
```

To deprecate a static method you must call deprecated\_function (**not** method), after the staticmethod call:

```
@staticmethod
@deprecated_function(deprecated_in((0, 1, 4)))
def create_repository(base, shared=False, format=None):
```

When you deprecate an API, you should not just delete its tests, because then we might introduce bugs in them. If the API is still present at all, it should still work. The basic approach is to use TestCase.applyDeprecated which in one step checks that the API gives the expected deprecation message, and also returns the real result from the method, so that tests can keep running.

Deprecation warnings will be suppressed for final releases, but not for development versions or release candidates, or when running bzr selftest. This gives developers information about whether their code is using deprecated functions, but avoids confusing users about things they can't fix.

## Getting Input

### **Processing Command Lines**

bzrlib has a standard framework for parsing command lines and calling processing routines associated with various commands. See builtins.py for numerous examples.

### Standard Parameter Types

There are some common requirements in the library: some parameters need to be unicode safe, some need byte strings, and so on. At the moment we have only codified one specific pattern: Parameters that need to be unicode should be checked via bzrlib.osutils.safe\_unicode. This will coerce the input into unicode in a consistent fashion, allowing trivial strings to be used for programmer convenience, but not performing unpredictably in the presence of different locales.

## Writing Output

(The strategy described here is what we want to get to, but it's not consistently followed in the code at the moment.)

bzrlib is intended to be a generically reusable library. It shouldn't write messages to stdout or stderr, because some programs that use it might want to display that information through a GUI or some other mechanism.

We can distinguish two types of output from the library:

1. Structured data representing the progress or result of an operation. For example, for a commit command this will be a list of the modified files and the finally committed revision number and id.

These should be exposed either through the return code or by calls to a callback parameter.

A special case of this is progress indicators for long-lived operations, where the caller should pass a ProgressBar object.

 Unstructured log/debug messages, mostly for the benefit of the developers or users trying to debug problems. This should always be sent through bzrlib.trace and Python logging, so that it can be redirected by the client.

The distinction between the two is a bit subjective, but in general if there is any chance that a library would want to see something as structured data, we should make it so.

The policy about how output is presented in the text-mode client should be only in the command-line tool.

### **Progress and Activity Indications**

bzrlib has a way for code to display to the user that stuff is happening during a long operation. There are two particular types: *activity* which means that IO is happening on a Transport, and *progress* which means that higher-level application work is occurring. Both are drawn together by the *ui\_factory*.

Transport objects are responsible for calling *report\_transport\_activity* when they do IO.

Progress uses a model/view pattern: application code acts on a *ProgressTask* object, which notifies the UI when it needs to be displayed. Progress tasks form a stack. To create a new progress task on top of the stack, call *bzrlib.ui.ui\_factory.nested\_progress\_bar()*, then call *update()* on the returned ProgressTask. It can be updated with just a text description, with a numeric count, or with a numeric count and expected total count. If an expected total count is provided the view can show the progress moving along towards the expected total.

The user should call *finish* on the *ProgressTask* when the logical operation has finished, so it can be removed from the stack.

Progress tasks have a complex relationship with generators: it's a very good place to use them, but because python2.4 does not allow finally blocks in generators it's hard to clean them up properly. In this case it's probably better to have the code calling the generator allocate a progress task for its use and then call *finalize* when it's done, which will close it if it was not already closed. The generator should also finish the progress task when it exits, because it may otherwise be a long time until the finally block runs.

https://wiki.ubuntu.com/UnitsPolicy provides a good explanation about which unit should be used when. Roughly speaking, IEC standard applies for base-2 units and SI standard applies for base-10 units:: \* for network bandwidth an disk sizes, use base-10 (Mbits/s, kB/s, GB), \* for RAM sizes, use base-2 (GiB, TiB).

## Displaying help

Bazaar has online help for various topics through bzr help COMMAND or equivalently bzr command -h. We also have help on command options, and on other help topics. (See help\_topics.py.)

As for python docstrings, the first paragraph should be a singlesentence synopsis of the command.

The help for options should be one or more proper sentences, starting with a capital letter and finishing with a full stop (period).

All help messages and documentation should have two spaces between sentences.

## Handling Errors and Exceptions

Commands should return non-zero when they encounter circumstances that the user should really pay attention to - which

includes trivial shell pipelines.

Recommended values are:

- 0. OK.
- 1. Conflicts in merge-like operations, or changes are present in diff-like operations.
- 2. Unrepresentable diff changes (i.e. binary files that we cannot show a diff of).
- 3. An error or exception has occurred.
- 4. An internal error occurred (one that shows a traceback.)

Errors are handled through Python exceptions. Exceptions should be defined inside bzrlib.errors, so that we can see the whole tree at a glance.

We broadly classify errors as either being either internal or not, depending on whether internal\_error is set or not. If we think it's our fault, we show a backtrace, an invitation to report the bug, and possibly other details. This is the default for errors that aren't specifically recognized as being caused by a user error. Otherwise we show a briefer message, unless -Derror was given.

Many errors originate as "environmental errors" which are raised by Python or builtin libraries – for example IOError. These are treated as being our fault, unless they're caught in a particular tight scope where we know that they indicate a user errors. For example if the repository format is not found, the user probably gave the wrong path or URL. But if one of the files inside the repository is not found, then it's our fault – either there's a bug in bzr, or something complicated has gone wrong in the environment that means one internal file was deleted.

Many errors are defined in <a href="https://bzrlib/errors.py">but it's OK for new</a> errors to be added near the place where they are used. Exceptions are formatted for the user by conversion to a string (eventually calling their <u>\_\_str\_\_</u> method.) As a convenience the .\_fmt member can be used as a template which will be mapped to the error's instance dict.

New exception classes should be defined when callers might want to catch that exception specifically, or when it needs a substantially different format string.

- 1. If it is something that a caller can recover from, a custom exception is reasonable.
- 2. If it is a data consistency issue, using a builtin like ValueError/TypeError is reasonable.
- 3. If it is a programmer error (using an api incorrectly) AssertionError is reasonable.
- 4. Otherwise, use BzrError Of InternalBzrError.

Exception strings should start with a capital letter and should not have a final fullstop. If long, they may contain newlines to break the text.

### Assertions

Do not use the Python assert statement, either in tests or elsewhere. A source test checks that it is not used. It is ok to explicitly raise AssertionError.

Rationale:

• It makes the behaviour vary depending on whether bzr is run with -O or not, therefore giving a chance for bugs that occur in one case or the other, several of which have already occurred: assertions with side effects, code which can't continue unless the assertion passes, cases where we should give the user a proper message rather than an assertion failure.

- It's not that much shorter than an explicit if/raise.
- It tends to lead to fuzzy thinking about whether the check is actually needed or not, and whether it's an internal error or not
- It tends to cause look-before-you-leap patterns.
- It's unsafe if the check is needed to protect the integrity of the user's data.
- It tends to give poor messages since the developer can get by with no explanatory text at all.
- We can't rely on people always running with -O in normal use, so we can't use it for tests that are actually expensive.
- Expensive checks that help developers are better turned on from the test suite or a -D flag.
- If used instead of self.assert\*() in tests it makes them falsely pass with -O.

## **Documenting Changes**

When you change bzrlib, please update the relevant documentation for the change you made: Changes to commands should update their help, and possibly end user tutorials; changes to the core library should be reflected in API documentation.

### NEWS File

If you make a user-visible change, please add a note to the NEWS file. The description should be written to make sense to someone who's just a user of bzr, not a developer: new functions or classes shouldn't be mentioned, but new commands, changes in behaviour or fixed nontrivial bugs should be listed. See the existing entries for an idea of what should be done.

Within each release, entries in the news file should have the most

user-visible changes first. So the order should be approximately:

- changes to existing behaviour the highest priority because the user's existing knowledge is incorrect
- new features should be brought to their attention
- bug fixes may be of interest if the bug was affecting them, and should include the bug number if any
- major documentation changes, including fixed documentation bugs
- changes to internal interfaces

People who made significant contributions to each change are listed in parenthesis. This can include reporting bugs (particularly with good details or reproduction recipes), submitting patches, etc.

To help with merging, NEWS entries should be sorted lexicographically within each section.

### Commands

The docstring of a command is used by bzr help to generate help output for the command. The list 'takes\_options' attribute on a command is used by bzr help to document the options for the command - the command docstring does not need to document them. Finally, the '\_see\_also' attribute on a command can be used to reference other related help topics.

### API Documentation

Functions, methods, classes and modules should have docstrings describing how they are used.

The first line of the docstring should be a self-contained sentence.

For the special case of Command classes, this acts as the user-

visible documentation shown by the help command.

The docstrings should be formatted as reStructuredText (like this document), suitable for processing using the epydoc tool into HTML documentation.

### **General Guidelines**

### Copyright

The copyright policy for bzr was recently made clear in this email (edited for grammatical correctness):

The attached patch cleans up the copyright and license statemen the bzr source. It also adds tests to help us remember to add t with the correct text.

We had the problem that lots of our files were "Copyright Canon Development Ltd" which is not a real company, and some other va on this theme. Also, some files were missing the GPL statements

I want to be clear about the intent of this patch, since copyri be a little controversial.

1) The big motivation for this is not to shut out the community just to clean up all of the invalid copyright statements.

2) It has been the general policy for bzr that we want a single copyright holder for all of the core code. This is following th set by the FSF, which makes it easier to update the code to a n license in case problems are encountered. (For example, if we w upgrade the project universally to GPL v3 it is much simpler if a single copyright holder). It also makes it clearer if copyrig ever debated, there is a single holder, which makes it easier t in court, etc. (I think the FSF position is that if you assign copyright, they can defend it in court rather than you needing I'm sure Canonical would do the same). As such, Canonical has requested copyright assignments from all

As such, Canonical has requested copyright assignments from all major contributers.

3) If someone wants to add code and not attribute it to Canonic is a specific list of files that are excluded from this check.

test failure indicates where that is, and how to update it.

4) If anyone feels that I changed a copyright statement incorre let me know, and I'll be happy to correct it. Whenever you have mechanical changes like this, it is possible to make some mista

Just to reiterate, this is a community project, and it is meant that way. Core bzr code is copyright Canonical for legal reason the tests are just there to help us maintain that.

**▲** 

Þ

## **Miscellaneous Topics**

## Debugging

Bazaar has a few facilities to help debug problems by going into pdb, the Python debugger.

If the **BZR\_PDB** environment variable is set then bzr will go into pdb post-mortem mode when an unhandled exception occurs.

If you send a SIGQUIT or SIGBREAK signal to bzr then it will drop into the debugger immediately. SIGQUIT can be generated by pressing Ctrl-\ on Unix. SIGBREAK is generated with Ctrl-Pause on Windows (some laptops have this as Fn-Pause). You can continue execution by typing c. This can be disabled if necessary by setting the environment variable BZR\_SIGQUIT\_PDB=0.

## Debug Flags

Bazaar accepts some global options starting with -D such as -Dhpss. These set a value in *bzrlib.debug.debug\_flags*, and typically cause more information to be written to the trace file. Most *mutter* calls should be guarded by a check of those flags so that we don't write out too much information if it's not needed.

Debug flags may have effects other than just emitting trace messages.

Run bzr help global-options to see them all.

These flags may also be set as a comma-separated list in the debug\_flags option in e.g. ~/.bazaar/bazaar.conf. (Note that it must be in this global file, not in the branch or location configuration,

because it's currently only loaded at startup time.) For instance you may want to always record hpss traces and to see full error tracebacks:

debug\_flags = hpss, error

### Jargon

#### revno

Integer identifier for a revision on the main line of a branch. Revision 0 is always the null revision; others are 1-based indexes into the branch's revision history.

## Unicode and Encoding Support

This section discusses various techniques that Bazaar uses to handle characters that are outside the ASCII set.

#### Command.outf

When a command object is created, it is given a member variable accessible by self.outf. This is a file-like object, which is bound to sys.stdout, and should be used to write information to the screen, rather than directly writing to sys.stdout or calling print. This file has the ability to translate Unicode objects into the correct representation, based on the console encoding. Also, the class attribute encoding\_type will effect how unprintable characters will be handled. This parameter can take one of 3 values:

replace

Unprintable characters will be represented with a suitable replacement marker (typically '?'), and no exception will be raised. This is for any command which generates text for the user to review, rather than for automated processing. For example: bzr log should not fail if one of the entries has text that cannot be displayed.

strict

Attempting to print an unprintable character will cause a UnicodeError. This is for commands that are intended more as scripting support, rather than plain user review. For example: bzr 1s is designed to be used with shell scripting. One use would be bzr 1s --null --unknowns | xargs -0 rm. If bzr printed a filename with a '?', the wrong file could be deleted. (At the very least, the correct file would not be deleted). An error is used to indicate that the requested action could not be performed.

exact

Do not attempt to automatically convert Unicode strings. This is used for commands that must handle conversion themselves. For example: bzr diff needs to translate Unicode paths, but should not change the exact text of the contents of the files.

bzrlib.urlutils.unescape\_for\_display

Because Transports work in URLs (as defined earlier), printing the raw URL to the user is usually less than optimal. Characters outside the standard set are printed as escapes, rather than the real character, and local paths would be printed as file:// urls. The function unescape\_for\_display attempts to unescape a URL, such that anything that cannot be printed in the current encoding stays an escaped URL, but valid characters are generated where possible.

## Portability Tips

The bzrlib.osutils module has many useful helper functions, including some more portable variants of functions in the standard

library.

In particular, don't use shutil.rmtree unless it's acceptable for it to fail on Windows if some files are readonly or still open elsewhere. Use bzrlib.osutils.rmtree instead.

## C Extension Modules

We write some extensions in C using pyrex. We design these to work in three scenarios:

- User with no C compiler
- User with C compiler
- Developers

The recommended way to install bzr is to have a C compiler so that the extensions can be built, but if no C compiler is present, the pure python versions we supply will work, though more slowly.

For developers we recommend that pyrex be installed, so that the C extensions can be changed if needed.

For the C extensions, the extension module should always match the original python one in all respects (modulo speed). This should be maintained over time.

To create an extension, add rules to setup.py for building it with pyrex, and with distutils. Now start with an empty .pyx file. At the top add "include 'yourmodule.py'". This will import the contents of foo.py into this file at build time - remember that only one module will be loaded at runtime. Now you can subclass classes, or replace functions, and only your changes need to be present in the .pyx file.

Note that pyrex does not support all 2.4 programming idioms, so some syntax changes may be required. I.e.

- 'from foo import (bar, gam)' needs to change to not use the brackets.
- 'import foo.bar as bar' needs to be 'import foo.bar; bar = foo.bar'

If the changes are too dramatic, consider maintaining the python code twice - once in the .pyx, and once in the .py, and no longer including the .py file.

## Making Installers for OS Windows

To build a win32 installer, see the instructions on the wiki page: http://bazaar-vcs.org/BzrWin32Installer

## Core Developer Tasks

## Overview

### What is a Core Developer?

While everyone in the Bazaar community is welcome and encouraged to propose and submit changes, a smaller team is reponsible for pulling those changes together into a cohesive whole. In addition to the general developer stuff covered above, "core" developers have responsibility for:

- reviewing changes
- reviewing blueprints
- planning releases
- managing releases (see Releasing Bazaar)

**Note:** Removing barriers to community participation is a key reason for adopting distributed VCS technology. While DVCS removes many technical barriers, a small number of social barriers are often necessary instead. By documenting how the above things are done, we hope to encourage more people to participate in these activities, keeping the differences between core and non-core contributors to a minimum.

### Communicating and Coordinating

While it has many advantages, one of the challenges of distributed development is keeping everyone else aware of what you're working on. There are numerous ways to do this:

- 1. Assign bugs to yourself in Launchpad
- 2. Mention it on the mailing list

3. Mention it on IRC

As well as the email notifcations that occur when merge requests are sent and reviewed, you can keep others informed of where you're spending your energy by emailing the **bazaar-commits** list implicitly. To do this, install and configure the Email plugin. One way to do this is add these configuration settings to your central configuration file (e.g. ~/.bazaar/bazaar.conf on Linux):

```
[DEFAULT]
email = Joe Smith <joe.smith@internode.on.net>
smtp_server = mail.internode.on.net:25
```

Then add these lines for the relevant branches in locations.conf:

```
post_commit_to = bazaar-commits@lists.canonical.com
post_commit_mailer = smtplib
```

While attending a sprint, RobertCollins' Dbus plugin is useful for the same reason. See the documentation within the plugin for information on how to set it up and configure it.

## Submitting Changes

### An Overview of PQM

Of the many workflows supported by Bazaar, the one adopted for Bazaar development itself is known as "Decentralized with automatic gatekeeper". To repeat the explanation of this given on http://bazaar-vcs.org/Workflows:

In this workflow, each developer has their own branch or branches, plus read-only access to the mainline. A software gatekeeper (e.g. PQM) has commit rights to the main branch. When a developer wants their work merged, they request the gatekeeper to merge it. The gatekeeper does a merge, a compile, and runs the test suite. If the code passes, it is merged into the mainline.

In a nutshell, here's the overall submission process:

- 1. get your work ready (including review except for trivial changes)
- 2. push to a public location
- 3. ask PQM to merge from that location

**Note:** At present, PQM always takes the changes to merge from a branch at a URL that can be read by it. For Bazaar, that means a public, typically http, URL.

As a result, the following things are needed to use PQM for submissions:

- 1. A publicly available web server
- 2. Your OpenPGP key registered with PQM (contact RobertCollins for this)
- 3. The PQM plugin installed and configured (not strictly required but highly recommended).

### Selecting a Public Branch Location

If you don't have your own web server running, branches can always be pushed to Launchpad. Here's the process for doing that:

Depending on your location throughout the world and the size of your repository though, it is often quicker to use an alternative public location to Launchpad, particularly if you can set up your own repo and push into that. By using an existing repo, push only needs to send the changes, instead of the complete repository every time. Note that it is easy to register branches in other locations with Launchpad so no benefits are lost by going this way. **Note:** For Canonical staff, http://people.ubuntu.com/~<user>/ is one suggestion for public http branches. Contact your manager for information on accessing this system if required.

It should also be noted that best practice in this area is subject to change as things evolve. For example, once the Bazaar smart server on Launchpad supports server-side branching, the performance situation will be very different to what it is now (Jun 2007).

Configuring the PQM Plug-In

While not strictly required, the PQM plugin automates a few things and reduces the chance of error. Before looking at the plugin, it helps to understand a little more how PQM operates. Basically, PQM requires an email indicating what you want it to do. The email typically looks like this:

star-merge source-branch target-branch

For example:

```
star-merge http://bzr.arbash-meinel.com/branches/bzr/jam-integr
```

Note that the command needs to be on one line. The subject of the email will be used for the commit message. The email also needs to be gpg signed with a key that PQM accepts.

The advantages of using the PQM plugin are:

- 1. You can use the config policies to make it easy to set up public branches, so you don't have to ever type the full paths you want to merge from or into.
- 2. It checks to make sure the public branch last revision matches the local last revision so you are submitting what you think you

are.

- 3. It uses the same public\_branch and smtp sending settings as bzr-email, so if you have one set up, you have the other mostly set up.
- 4. Thunderbird refuses to not wrap lines, and request lines are usually pretty long (you have 2 long URLs in there).

Here are sample configuration settings for the PQM plugin. Here are the lines in bazaar.conf:

```
[DEFAULT]
email = Joe Smith <joe.smith@internode.on.net>
smtp_server=mail.internode.on.net:25
```

And here are the lines in locations.conf (Or branch.conf for dirstatetags branches):

```
[/home/joe/bzr/my-integration]
push_location = sftp://joe-smith@bazaar.launchpad.net/%7Ejoe-sm
push_location:policy = norecurse
public_branch = http://bazaar.launchpad.net/~joe-smith/bzr/my-i
public_branch:policy = appendpath
pqm_email = Bazaar PQM <pqm@bazaar-vcs.org>
pqm_branch = http://bazaar-vcs.org/bzr/bzr.dev
```

Note that the push settings will be added by the first push on a branch. Indeed the preferred way to generate the lines above is to use push with an argument, then copy-and-paste the other lines into the relevant file.

### Submitting a Change

Here is one possible recipe once the above environment is set up:

- 1. pull bzr.dev => my-integration
- 2. merge patch => my-integration

- 3. fix up any final merge conflicts (NEWS being the big killer here).
- 4. commit
- 5. push
- 6. pqm-submit

**Note:** The push step is not required if my-integration is a checkout of a public branch.

Because of defaults, you can type a single message into commit and pqm-commit will reuse that.

### Tracking Change Acceptance

The web interface to PQM is https://pqm.bazaar-vcs.org/. After submitting a change, you can visit this URL to confirm it was received and placed in PQM's queue.

When PQM completes processing a change, an email is sent to you with the results.

## **Reviewing Blueprints**

### **Blueprint Tracking Using Launchpad**

New features typically require a fair amount of discussion, design and debate. For Bazaar, that information is often captured in a socalled "blueprint" on our Wiki. Overall tracking of blueprints and their status is done using Launchpad's relevant tracker, https://blueprints.launchpad.net/bzr/. Once a blueprint for ready for review, please announce it on the mailing list.

Alternatively, send an email beginning with [RFC] with the proposal to the list. In some cases, you may wish to attach proposed code or a proposed developer document if that best communicates the idea. Debate can then proceed using the normal merge review processes.

Recording Blueprint Review Feedback

Unlike its Bug Tracker, Launchpad's Blueprint Tracker doesn't currently (Jun 2007) support a chronological list of comment responses. Review feedback can either be recorded on the Wiki hosting the blueprints or by using Launchpad's whiteboard feature.

## Planning Releases

Using Releases and Milestones in Launchpad

TODO ... (Exact policies still under discussion)

### **Bug Triage**

Keeping on top of bugs reported is an important part of ongoing release planning. Everyone in the community is welcome and encouraged to raise bugs, confirm bugs raised by others, and nominate a priority. Practically though, a good percentage of bug triage is often done by the core developers, partially because of their depth of product knowledge.

With respect to bug triage, core developers are encouraged to play an active role with particular attention to the following tasks:

- keeping the number of unconfirmed bugs low
- ensuring the priorities are generally right (everything as critical or medium - is meaningless)
- looking out for regressions and turning those around sooner rather than later.

**Note:** As well as prioritizing bugs and nominating them against a target milestone, Launchpad lets core developers offer to mentor

others in fixing them.

 $\bullet$  Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

In the second se

# Bazaar Testing Guide

## The Importance of Testing

Reliability is a critical success factor for any Version Control System. We want Bazaar to be highly reliable across multiple platforms while evolving over time to meet the needs of its community.

In a nutshell, this is what we expect and encourage:

• New functionality should have test cases. Preferably write the test before writing the code.

In general, you can test at either the command-line level or the internal API level. See Writing tests below for more detail.

• Try to practice Test-Driven Development: before fixing a bug, write a test case so that it does not regress. Similarly for adding a new feature: write a test case for a small version of the new feature before starting on the code itself. Check the test fails on the old code, then add the feature or fix and check it passes.

By doing these things, the Bazaar team gets increased confidence that changes do what they claim to do, whether provided by the core team or by community members. Equally importantly, we can be surer that changes down the track do not break new features or bug fixes that you are contributing today.

As of September 2009, Bazaar ships with a test suite containing over 23,000 tests and growing. We are proud of it and want to remain so. As community members, we all benefit from it. Would you trust version control on your project to a product *without* a test suite like Bazaar has?

## Running the Test Suite

As of Bazaar 2.1, you must have the testtools library installed to run the bzr test suite.

To test all of Bazaar, just run:

bzr selftest

With --verbose bzr will print the name of every test as it is run.

This should always pass, whether run from a source tree or an installed copy of Bazaar. Please investigate and/or report any failures.

### Running particular tests

Currently, bzr selftest is used to invoke tests. You can provide a pattern argument to run a subset. For example, to run just the blackbox tests, run:

```
./bzr selftest -v blackbox
```

To skip a particular test (or set of tests), use the –exclude option (shorthand -x) like so:

./bzr selftest -v -x blackbox

To ensure that all tests are being run and succeeding, you can use the –strict option which will fail if there are any missing features or known failures, like so:

./bzr selftest --strict

To list tests without running them, use the –list-only option like so:

./bzr selftest --list-only

This option can be combined with other selftest options (like -x) and filter patterns to understand their effect.

Once you understand how to create a list of tests, you can use the – load-list option to run only a restricted set of tests that you kept in a file, one test id by line. Keep in mind that this will never be sufficient to validate your modifications, you still need to run the full test suite for that, but using it can help in some cases (like running only the failed tests for some time):

./bzr selftest -- load-list my\_failing\_tests

This option can also be combined with other selftest options, including patterns. It has some drawbacks though, the list can become out of date pretty quick when doing Test Driven Development.

To address this concern, there is another way to run a restricted set of tests: the –starting-with option will run only the tests whose name starts with the specified string. It will also avoid loading the other tests and as a consequence starts running your tests quicker:

```
./bzr selftest --starting-with bzrlib.blackbox
```

This option can be combined with all the other selftest options including –load-list. The later is rarely used but allows to run a subset of a list of failing tests for example.

### Disabling plugins

To test only the bzr core, ignoring any plugins you may have

installed, use:

./bzr --no-plugins selftest

## Disabling crash reporting

By default Bazaar uses apport to report program crashes. In developing Bazaar it's normal and expected to have it crash from time to time, at least because a test failed if for no other reason.

Therefore you should probably add debug\_flags = no\_apport to your bazaar.conf file (in ~/.bazaar/ on Unix), so that failures just print a traceback rather than writing a crash file.

### Test suite debug flags

Similar to the global -Dfoo debug options, bzr selftest accepts -E=foo debug flags. These flags are:

do not clear the global debug flags when running a test. This can provide useful logging to help debug test failures when used with e.g. bzr -Dhpss allow\_debug: selftest -E=allow\_debug Note that this will probably cause some tests to fail, because they don't expect to run with any debug flags on.

### Using subunit

Bazaar can optionally produce output in the machine-readable subunit format, so that test output can be post-processed by various tools. To generate a subunit test stream:

```
$ ./bzr selftest --subunit
```

Processing such a stream can be done using a variety of tools including:

- The builtin subunit2pyunit, subunit-filter, subunit-ls, subunit2junitxml from the subunit project.
- tribunal, a GUI for showing test results.
- testrepository, a tool for gathering and managing test runs.

## Using testrepository

Bazaar ships with a config file for testrepository. This can be very useful for keeping track of failing tests and doing general workflow support. To run tests using testrepository:

\$ testr run

To run only failing tests:

```
$ testr run --failing
```

To run only some tests, without plugins:

```
$ test run test_selftest -- --no-plugins
```

See the testrepository documentation for more details.

## Writing Tests

Normally you should add or update a test for all bug fixes or new features in Bazaar.

## Where should I put a new test?

Bzrlib's tests are organised by the type of test. Most of the tests in bzr's test suite belong to one of these categories:

- Unit tests
- Blackbox (UI) tests
- Per-implementation tests
- Doctests

A quick description of these test types and where they belong in bzrlib's source follows. Not all tests fall neatly into one of these categories; in those cases use your judgement.

#### Unit tests

Unit tests make up the bulk of our test suite. These are tests that are focused on exercising a single, specific unit of the code as directly as possible. Each unit test is generally fairly short and runs very quickly.

They are found in <a href="mailto:bzrlib/tests/test\_\*.py">bzrlib/tests/test\_\*.py</a>. So in general tests should be placed in a file named test\_FOO.py where FOO is the logical thing under test.

For example, tests for merge3 in bzrlib belong in bzrlib/tests/test\_merge3.py. See bzrlib/tests/test\_sampler.py for a template test script.

### Blackbox (UI) tests

Tests can be written for the UI or for individual areas of the library. Choose whichever is appropriate: if adding a new command, or a new command option, then you should be writing a UI test. If you are both adding UI functionality and library functionality, you will want to write tests for both the UI and the core behaviours. We call UI tests 'blackbox' tests and they belong in bzrlib/tests/blackbox/\*.py.

When writing blackbox tests please honour the following conventions:

- 1. Place the tests for the command 'name' in bzrlib/tests/blackbox/test\_name.py. This makes it easy for developers to locate the test script for a faulty command.
- 2. Use the 'self.run\_bzr("name")' utility function to invoke the command rather than running bzr in a subprocess or invoking the cmd\_object.run() method directly. This is a lot faster than subprocesses and generates the same logging output as running it in a subprocess (which invoking the method directly does not).
- 3. Only test the one command in a single test script. Use the bzrlib library when setting up tests and when evaluating the side-effects of the command. We do this so that the library api has continual pressure on it to be as functional as the command line in a simple manner, and to isolate knock-on effects throughout the blackbox test suite when a command changes its name or signature. Ideally only the tests for a given command are affected when a given command is changed.
- If you have a test which does actually require running bzr in a subprocess you can use run\_bzr\_subprocess. By default the spawned process will not load plugins unless --allowplugins is supplied.
Per-implementation tests are tests that are defined once and then run against multiple implementations of an interface. For example, per\_transport.py defines tests that all Transport implementations (local filesystem, HTTP, and so on) must pass. They are found in bzrlib/tests/per\_\*/\*.py, and bzrlib/tests/per\_\*.py.

These are really a sub-category of unit tests, but an important one.

Along the same lines are tests for extension modules. We generally have both a pure-python and a compiled implementation for each module. As such, we want to run the same tests against both implementations. These can generally be found in bzrlib/tests/\*\_\_\*.py since extension modules are usually prefixed with an underscore. Since there are only two implementations, we have a helper function bzrlib.tests.permute\_for\_extension, which can simplify the load\_tests implementation.

#### Doctests

We make selective use of doctests. In general they should provide *examples* within the API documentation which can incidentally be tested. We don't try to test every important case using doctests — regular Python tests are generally a better solution. That is, we just use doctests to make our documentation testable, rather than as a way to make tests.

Most of these are in bzrlib/doc/api. More additions are welcome.

### Shell-like tests

bzrlib/tests/script.py allows users to write tests in a syntax very close to a shell session, using a restricted and limited set of commands that should be enough to mimic most of the behaviours.

A script is a set of commands, each command is composed of:

- one mandatory command line,
- one optional set of input lines to feed the command,
- one optional set of output expected lines,
- one optional set of error expected lines.

Input, output and error lines can be specified in any order.

Except for the expected output, all lines start with a special string (based on their origin when used under a Unix shell):

- '\$ ' for the command,
- '<' for input,
- nothing for output,
- '2>' for errors,

Comments can be added anywhere, they start with '#' and end with the line.

The execution stops as soon as an expected output or an expected error is not matched.

When no output is specified, any ouput from the command is accepted and execution continue.

If an error occurs and no expected error is specified, the execution stops.

An error is defined by a returned status different from zero, not by the presence of text on the error stream.

The matching is done on a full string comparison basis unless '...' is used, in which case expected output/errors can be less precise.

Examples:

The following will succeeds only if 'bzr add' outputs 'adding file':

\$ bzr add file
>adding file

If you want the command to succeed for any output, just use:

\$ bzr add file

The following will stop with an error:

\$ bzr not-a-command

If you want it to succeed, use:

\$ bzr not-a-command 2> bzr: ERROR: unknown command "not-a-command"

You can use ellipsis (...) to replace any piece of text you don't want to be matched exactly:

```
$ bzr branch not-a-branch
2>bzr: ERROR: Not a branch...not-a-branch/".
```

This can be used to ignore entire lines too:

```
$ cat
<first line
<second line
<third line
# And here we explain that surprising fourth line
<fourth line
<last line
>first line
>...
>last line
```

You can check the content of a file with cat:

```
$ cat <file
>expected content
```

You can also check the existence of a file with cat, the following will fail if the file doesn't exist:

\$ cat file

The actual use of ScriptRunner within a TestCase looks something like this:

```
def test_unshelve_keep(self):
    # some setup here
    sr = ScriptRunner()
    sr.run_script(self, '''
$ bzr add file
$ bzr shelve --all -m Foo
$ bzr shelve --list
1: Foo
$ bzr unshelve --keep
$ bzr shelve --list
1: Foo
$ cat file
contents of file
''')
```

### Import tariff tests

*bzrlib.tests.test\_import\_tariff* has some tests that measure how many Python modules are loaded to run some representative commands.

We want to avoid loading code unnecessarily, for reasons including:

- Python modules are interpreted when they're loaded, either to define classes or modules or perhaps to initialize some structures.
- With a cold cache we may incur blocking real disk IO for each module.
- Some modules depend on many others.

• Some optional modules such as *testtools* are meant to be soft dependencies and only needed for particular cases. If they're loaded in other cases then bzr may break for people who don't have those modules.

*test\_import\_tarrif* allows us to check that removal of imports doesn't regress.

This is done by running the command in a subprocess with -profile-imports. Starting a whole Python interpreter is pretty slow, so we don't want exhaustive testing here, but just enough to guard against distinct fixed problems.

Assertions about precisely what is loaded tend to be brittle so we instead make assertions that particular things aren't loaded.

Unless selftest is run with --no-plugins, modules will be loaded in the usual way and checks made on what they cause to be loaded. This is probably worth checking into, because many bzr users have at least some plugins installed (and they're included in binary installers).

In theory, plugins might have a good reason to load almost anything: someone might write a plugin that opens a network connection or pops up a gui window every time you run 'bzr status'. However, it's more likely that the code to do these things is just being loaded accidentally. We might eventually need to have a way to make exceptions for particular plugins.

Some things to check:

- non-GUI commands shouldn't load GUI libraries
- operations on bzr native formats sholudn't load foreign branch libraries
- network code shouldn't be loaded for purely local operations

• particularly expensive Python built-in modules shouldn't be loaded unless there is a good reason

### Testing locking behaviour

You may want to write tests that particular objects are or aren't locked during particular operations: see for example bug 498409.

The *TestCase* base class registers hooks that record lock actions into .\_lock\_actions in this format:

4		•
	('released',	<pre>LockResult(file:///tmp/testbzr-J2pcy2.tmp/.bzr/b</pre>
	('acquired',	<pre>LockResult(file:///tmp/testbzr-J2pcy2.tmp/.bzr/b</pre>
	('released',	<pre>LockResult(file:///tmp/testbzr-J2pcy2.tmp/.bzr/r</pre>
	('acquired',	<pre>LockResult(file:///tmp/testbzr-J2pcy2.tmp/.bzr/r</pre>
	('released',	<pre>LockResult(file:///tmp/testbzr-J2pcy2.tmp/.bzr/b</pre>
	('acquired',	<pre>LockResult(file:///tmp/testbzr-J2pcy2.tmp/.bzr/b</pre>
Γ		

Alternatively you can register your own hooks to make custom assertions: see *TestCase.\_check\_locks* for an example.

### Skipping tests

In our enhancements to unittest we allow for some addition results beyond just success or failure.

If a test can't be run, it can say that it's skipped by raising a special exception. This is typically used in parameterized tests — for example if a transport doesn't support setting permissions, we'll skip the tests that relating to that.

```
try:
    return self.branch_format.initialize(repo.bzrdir)
except errors.UninitializableFormat:
    raise tests.TestSkipped('Uninitializable branch format')
```

Raising TestSkipped is a good idea when you want to make it clear that the test was not run, rather than just returning which makes it look as if it was run and passed.

Several different cases are distinguished:

#### TestSkipped

Generic skip; the only type that was present up to bzr 0.18.

TestNotApplicable

The test doesn't apply to the parameters with which it was run. This is typically used when the test is being applied to all implementations of an interface, but some aspects of the interface are optional and not present in particular concrete implementations. (Some tests that should raise this currently either silently return or raise TestSkipped.) Another option is to use more precise parameterization to avoid generating the test at all.

#### UnavailableFeature

The test can't be run because a dependency (typically a Python library) is not available in the test environment. These are in general things that the person running the test could fix by installing the library. It's OK if some of these occur when an end user runs the tests or if we're specifically testing in a limited environment, but a full test should never see them.

See Test feature dependencies below.

#### KnownFailure

The test exists but is known to fail, for example this might be appropriate to raise if you've committed a test for a bug but not the fix for it, or if something works on Unix but not on Windows.

Raising this allows you to distinguish these failures from the ones that are not expected to fail. If the test would fail because of something we don't expect or intend to fix, KnownFailure is not appropriate, and TestNotApplicable might be better.

KnownFailure should be used with care as we don't want a proliferation of quietly broken tests.

We plan to support three modes for running the test suite to control the interpretation of these results. Strict mode is for use in situations like merges to the mainline and releases where we want to make sure that everything that can be tested has been tested. Lax mode is for use by developers who want to temporarily tolerate some known failures. The default behaviour is obtained by bzr selftest with no options, and also (if possible) by running under another unittest harness.

result	strict	default	lax
TestSkipped	pass	pass	pass
TestNotApplicable	pass	pass	pass
UnavailableFeature	fail	pass	pass
KnownFailure	fail	pass	pass

### Test feature dependencies

#### Writing tests that require a feature

Rather than manually checking the environment in each test, a test class can declare its dependence on some test features. The feature objects are checked only once for each run of the whole test suite.

(For historical reasons, as of May 2007 many cases that should depend on features currently raise TestSkipped.)

For example:

```
class TestStrace(TestCaseWithTransport):
```

```
_test_needs_features = [StraceFeature]
```

This means all tests in this class need the feature. If the feature is not available the test will be skipped using UnavailableFeature.

Individual tests can also require a feature using the requireFeature method:

```
self.requireFeature(StraceFeature)
```

The old naming style for features is CamelCase, but because they're actually instances not classses they're now given instance-style names like apport.

Features already defined in bzrlib.tests and bzrlib.tests.features include:

- apport
- paramiko
- SymlinkFeature
- HardlinkFeature
- OsFifoFeature
- UnicodeFilenameFeature
- FTPServerFeature
- CaseInsensitiveFilesystemFeature.
- chown\_feature: The test can rely on OS being POSIX and python supporting os.chown.
- posix\_permissions\_feature: The test can use POSIX-style user/group/other permission bits.

#### Defining a new feature that tests can require

New features for use with \_test\_needs\_features Or requireFeature are defined by subclassing bzrlib.tests.Feature and overriding the \_probe and feature\_name methods. For example:

```
class _SymlinkFeature(Feature):
```

```
def _probe(self):
    return osutils.has_symlinks()
    def feature_name(self):
        return 'symlinks'
SymlinkFeature = _SymlinkFeature()
```

A helper for handling running tests based on whether a python module is available. This can handle 3rd-party dependencies (is paramiko available?) as well as stdlib (termios) or extension modules (bzrlib.\_groupcompress\_pyx). You create a new feature instance with:

```
# in bzrlib/tests/features.py
apport = tests.ModuleAvailableFeature('apport')
# then in bzrlib/tests/test_apport.py
class TestApportReporting(TestCaseInTempDir):
    _test_needs_features = [features.apport]
```

### Testing exceptions and errors

It's important to test handling of errors and exceptions. Because this code is often not hit in ad-hoc testing it can often have hidden bugs – it's particularly common to get NameError because the exception code references a variable that has since been renamed.

In general we want to test errors at two levels:

 A test in test\_errors.py checking that when the exception object is constructed with known parameters it produces an expected string form. This guards against mistakes in writing the format string, or in the str representations of its parameters. There should be one for each exception class. 2. Tests that when an api is called in a particular situation, it raises an error of the expected class. You should typically use assertRaises, which in the Bazaar test suite returns the exception object to allow you to examine its parameters.

In some cases blackbox tests will also want to check error reporting. But it can be difficult to provoke every error through the commandline interface, so those tests are only done as needed eg in response to a particular bug or if the error is reported in an unusual way(?) Blackbox tests should mostly be testing how the command-line interface works, so should only test errors if there is something particular to the cli in how they're displayed or handled.

### Testing warnings

The Python warnings module is used to indicate a non-fatal code problem. Code that's expected to raise a warning can be tested through callCatchWarnings.

The test suite can be run with -werror to check no unexpected errors occur.

However, warnings should be used with discretion. It's not an appropriate way to give messages to the user, because the warning is normally shown only once per source line that causes the problem. You should also think about whether the warning is serious enought that it should be visible to users who may not be able to fix it.

### Interface implementation testing and test scenarios

There are several cases in Bazaar of multiple implementations of a common conceptual interface. ("Conceptual" because it's not necessary for all the implementations to share a base class, though

they often do.) Examples include transports and the working tree, branch and repository classes.

In these cases we want to make sure that every implementation correctly fulfils the interface requirements. For example, every Transport should support the has() and get() and clone() methods. We have a sub-suite of tests in test\_transport\_implementations. (Most per-implementation tests are in submodules of bzrlib.tests, but not the transport tests at the moment.)

These tests are repeated for each registered Transport, by generating a new TestCase instance for the cross product of test methods and transport implementations. As each test runs, it has transport\_class and transport\_server set to the class it should test. Most tests don't access these directly, but rather use self.get\_transport which returns a transport of the appropriate type.

The goal is to run per-implementation only the tests that relate to that particular interface. Sometimes we discover a bug elsewhere that happens with only one particular transport. Once it's isolated, we can consider whether a test should be added for that particular implementation, or for all implementations of the interface.

The multiplication of tests for different implementations is normally accomplished by overriding the load\_tests function used to load tests from a module. This function typically loads all the tests, then applies a TestProviderAdapter to them, which generates a longer suite containing all the test variations.

See also Per-implementation tests (above).

#### Test scenarios

Some utilities are provided for generating variations of tests. This

can be used for per-implementation tests, or other cases where the same test code needs to run several times on different scenarios.

The general approach is to define a class that provides test methods, which depend on attributes of the test object being pre-set with the values to which the test should be applied. The test suite should then also provide a list of scenarios in which to run the tests.

Typically multiply\_tests\_from\_modules should be called from the test module's load\_tests function.

#### Test support

We have a rich collection of tools to support writing tests. Please use them in preference to ad-hoc solutions as they provide portability and performance benefits.

#### TestCase and its subclasses

The bzrlib.tests module defines many TestCase classes to help you write your tests.

#### TestCase

A base TestCase that extends the Python standard library's TestCase in several ways. It adds more assertion methods (e.g. assertContainsRe), addCleanup, and other features (see its API docs for details). It also has a setup that makes sure that global state like registered hooks and loggers won't interfere with your test. All tests should use this base class (whether directly or via a subclass).

#### TestCaseWithMemoryTransport

Extends TestCase and adds methods like get\_transport, make\_branch and make\_branch\_builder. The files created are stored in a MemoryTransport that is discarded at the end of the test. This class is good for tests that need to make branches or use transports, but that don't require storing things on disk. All tests that create bzrdirs should use this base class (either directly or via a subclass) as it ensures that the test won't accidentally operate on real branches in your filesystem.

TestCaseInTempDir

Extends TestCaseWithMemoryTransport. For tests that really do need files to be stored on disk, e.g. because a subprocess uses a file, or for testing functionality that accesses the filesystem directly rather than via the Transport layer (such as dirstate).

TestCaseWithTransport

Extends TestCaseInTempDir. Provides get\_url and get\_readonly\_url facilities. Subclasses can control the transports used by setting vfs\_transport\_factory, transport\_server and/or transport\_readonly\_server.

See the API docs for more details.

#### BranchBuilder

When writing a test for a feature, it is often necessary to set up a branch with a certain history. The BranchBuilder interface allows the creation of test branches in a quick and easy manner. Here's a sample session:

```
builder = self.make_branch_builder('relpath')
builder.build_commit()
builder.build_commit()
builder.build_commit()
branch = builder.get_branch()
```

make\_branch\_builder is a method of TestCaseWithMemoryTransport.

Note that many current tests create test branches by inheriting from TestCaseWithTransport and using the make\_branch\_and\_tree helper

to give them a WorkingTree that they can commit to. However, using the newer make\_branch\_builder helper is preferred, because it can build the changes in memory, rather than on disk. Tests that are explicitly testing how we work with disk objects should, of course, use a real WorkingTree.

Please see bzrlib.branchbuilder for more details.

If you're going to examine the commit timestamps e.g. in a test for log output, you should set the timestamp on the tree, rather than using fuzzy matches in the test.

#### TreeBuilder

The TreeBuilder interface allows the construction of arbitrary trees with a declarative interface. A sample session might look like:

```
tree = self.make_branch_and_tree('path')
builder = TreeBuilder()
builder.start_tree(tree)
builder.build(['foo', "bar/", "bar/file"])
tree.commit('commit the tree')
builder.finish_tree()
```

Usually a test will create a tree using make\_branch\_and\_memory\_tree
(a method of TestCaseWithMemoryTransport) Or make\_branch\_and\_tree
(a method of TestCaseWithTransport).

Please see bzrlib.treebuilder for more details.

igodolmollimits Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

In the second se

# **Releasing Bazaar**

This document describes the processes for making and announcing a Bazaar release, and managing the release process. This is just one phase of the overall development cycle, but it's the most complex part. This document gives a checklist you can follow from start to end in one go.

If you're helping the Release Manager (RM) for one reason or another, you may notice that he didn't follow that document scrupulously. He may have good reasons to do that but he may also have missed some parts.

Follow the document yourself and don't hesitate to create the missing milestones for example (we tend to forget these ones a lot).

trunk

Contents				
• Reie 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	asing Bazaar Preconditions Starting a cycle Starting the release phase Making the source tarball Publishing the source tarball Announcing the source freeze Publishing the release Announcing the release Merging the released code back to Releases until the final one See also			
	Starting a cycle Starting the release phase Making the source tarball Publishing the source tarball Announcing the source freeze Publishing the release Announcing the release Merging the released code back Releases until the final one See also			

## Preconditions

1. Download the pqm plugin and install it into your ~/.bazaar/plugins:

bzr branch lp:bzr-pqm ~/.bazaar/plugins/pqm

## Starting a cycle

To start a new release cycle:

- 1. Create a new series at <<u>https://launchpad.net/bzr/+addseries</u>>. There is one series for every *x.y* release.
- 2. Go to the series web page at <https://launchpad.net/bzr/x.y>
- 3. Create a new release at <<u>https://launchpad.net/bzr/x.y/+addrelease></u> and add information about this release. We will not use it yet, but it will be available for targeting or nominating bugs.
- 4. We create a new pqm-controlled branch for this release series, by asking a Canonical sysadmin. This branch means that from the first release beta or candidate onwards, general development continues on the trunk, and only specificallytargeted fixes go into the release branch.
- 5. Add milestones at <<u>https://edge.launchpad.net/bzr/x.y/+addmilestone></u> to that series for the beta release, release candidate and the final release, and their expected dates.
- 6. Update the version number in the bzr script, and the bzrlib/\_\_init\_\_.py file. Make sure there is always a corresponding milestone when you change that version number.
- Add a new section at the top of NEWS about the new release, including its version number and the headings from NEWStemplate.txt.
- 8. Send mail to the list with the key dates, who will be the release

manager, and the main themes or targeted bugs. Ask people to nominate objectives, or point out any high-risk things that are best done early, or that interact with other changes. This is called the metronome mail and is described in Development cycles.

9. Make a local branch for preparing this release. (Only for the first release in a series, otherwise you should already have a branch.)

```
bzr branch trunk prepare-1.14
```

10. Configure pqm-submit for this branch, with a section like this (where x.y is the version to release). ~/.bazaar/locations.conf:

```
[/home/mbp/bzr/prepare-x.y]
pqm_email = Canonical PQM <pqm@bazaar-vcs.org>
submit_branch = http://bazaar.launchpad.net/~bzr-pqm/bz
parent_branch = http://bazaar.launchpad.net/~bzr-pqm/bz
public_branch = http://bazaar.example.com/prepare-x.y
submit_to = bazaar@lists.canonical.com
smtp_server = mail.example.com:25
Please see <http://doc.bazaar-vcs.org/developers/HACKING.ht
for more details on PQM
</pre>
```

11. In the release branch, update version\_info in ./bzrlib/\_\_init\_\_.py. Make sure the corresponding milestone exists. Double check that ./bzr \_script\_version matches version\_info. Check the output of bzr --version.

For beta releases use:

version\_info = (2, 1, 0, 'beta', 1)

For release candidates use:

version\_info = (2, 0, 1, 'candidate', 1)

## Starting the release phase

- 1. Create a new milestone at <<u>https://launchpad.net/bzr/x.y/+addmilestone></u> for the beta release or release candidate if you haven't already.
- 2. Add the date and release number to ./NEWS

Depending on whether you're doing a beta or a bugfix release, you'll have to create a NEWS section for your release in the right place. Most of the time, the new section is at the top of the file (look what have been done for the various 2.0x and 2.1.0bx releases). The rule is to keep the sections sorted by date. You'll need to be cautious when merging back to trunk to respect that.

3. To check that all bugs mentioned in ./NEWS are actually marked as closed in Launchpad, you can run tools/check-newsbugs.py:

./tools/check-newsbugs.py NEWS

(But note there can be some false positives, and this script may be flaky <<u>https://bugs.edge.launchpad.net/bzr/+bug/354985</u>>. Don't let this slow you down too much.)

- 4. Summarize into one or two paragraphs what's new in this release.
- 5. Commit these changes to the release branch, using a command like:

```
bzr commit -m "Release 1.14."
```

The diff before you commit will be something like:

```
=== modified file 'NEWS'
--- NFWS
               2008-09-17 23:09:18 +0000
+++ NEWS
              2008-09-23 16:14:54 +0000
@@ -4,6 +4,23 @@
 .. contents::
+bzr 1.7 2008-09-23
+-----
+This release includes many bug fixes and a few performance
+improvements. ``bzr rm`` will now scan for missing files
+like how ``bzr add`` scans for unknown files and adds them
+polish has been applied to the stacking code. The b-tree i
+been brought in, with an eye on using it in a future repos
+There are only minor installer changes since bzr-1.7rc2.
+
 bzr 1.7rc2 2008-09-17
     . . . . . . . . . . . . . . . . . . .
=== modified file 'bzrlib/__init__.py'
--- bzrlib/__init__.py 2008-09-16 21:39:28 +0000
+++ bzrlib/__init__.py 2008-09-23 16:14:54 +0000
@@ -41,7 +41,7 @@
 # Python version 2.0 is (2, 0, 0, 'final', 0)." Additiona
 # releaselevel of 'dev' for unreleased under-development c
-version_info = (1, 7, 0, 'candidate', 2)
+version_info = (1, 7, 0, 'final', 0)
 # API compatibility version: bzrlib is currently API compa
•
```

6. Tag the new release:

bzr tag bzr-1.14

7. Push those changes to a bzr reposistory that is public and accessible on the Internet. PQM will pull from this repository when it attempts to merge your changes. Then submit those changes to PQM for merge into the appropriate release branch:

```
bzr push
bzr pqm-submit -m "(mbp) prepare 1.14"
```

8. When PQM succeeds, pull down the master release branch.

## Making the source tarball

1. Change into the source directory and run

make dist

2. Now we'll try expanding this tarball and running the test suite to check for packaging problems:

make check-dist-tarball

You may encounter failures while running the test suite caused by your locally installed plugins. Use your own judgment to decide if you can release with these failures. When in doubt, disable the faulty plugins one by one until you get no more failures.

## Publishing the source tarball

- 1. Go to the relevant milestone page in Launchpad.
- 2. Within that release, upload the source tarball and the GPG signature. Or, if you prefer, use the tools/packaging/lp-upload-release script to do this.

## Announcing the source freeze

1. Post to the bazaar list, saying that the source has been frozen. This is the cue for platform maintainers and plugin authors to update their code. This is done before the general public announcement of the release.

## Publishing the release

There is normally a delay of a few days after the source freeze to allow for binaries to be built on various platforms. Once they have been built, we have a releasable product. The next step is to make it generally available to the world.

go to the release

- 1. Within that release, upload the source tarball and zipfile and the GPG signature. Or, if you prefer, use the tools/packaging/lp-upload-release script to do this.
- 2. Link from http://bazaar-vcs.org/SourceDownloads to the tarball and signature.
- 3. Announce on the Bazaar website. This page is edited via the lp:bzr-website branch. (Changes pushed to this branch are refreshed by a cron job on escudero.)
- 4. Announce on the Bazaar wiki.
- Check that the documentation for this release is available in <http://doc.bazaar-vcs.org>. It should be automatically build when the branch is created, by a cron script update-bzr-docs on escudero. As of today (2009-08-27) igc manually updates the pretty version of it.

## Announcing the release

Now that the release is publicly available, tell people about it.

1. Make an announcement mail.

For release candidates or beta releases, this is sent to the bazaar list only to inform plugin authors and package or installer managers.

Once the installers are available, the mail can be sent to the bazaar-announce list too.

For final releases, it should also be cc'd to info-gnu@gnu.org, python-announce-list@python.org, bug-directory@gnu.org.

In all cases, it is good to set Reply-To: bazaar@lists.canonical.com, so that people who reply to the announcement don't spam other lists.

The announce mail will look something like this:

Subject: bzr x.yy released!

<<Summary paragraph from news>>

The Bazaar team is happy to announce availability of a new release of the bzr adaptive version control system. Bazaar is part of the GNU system <a href="http://gnu.org/">http://gnu.org/</a>>.

Thanks to everyone who contributed patches, suggestions, an feedback.

Bazaar is now available for download from http://bazaar-vcs.org/Download as a source tarball; package for various systems will be available soon.

<<NEWS section from this release back to the last major rel

<u>, ↓]</u>

Feel free to tweak this to your taste.

- 2. Make an announcement through <a href="https://launchpad.net/bzr/+announce">https://launchpad.net/bzr/+announce</a>
- 3. Update the IRC channel topic. Use the /topic command to do this, ensuring the new topic text keeps the project name, web site link, etc.
- 4. Announce on http://freshmeat.net/projects/bzr/

This should be done for beta releases, release candidates and final releases. If you do not have a Freshmeat account yet, ask one of the existing admins.

- Update http://en.wikipedia.org/wiki/Bazaar\_(software) this should be done for final releases but not for beta releases or Release Candidates.
- 6. Update the python package index: <a href="http://pypi.python.org/pypi/bzr">http://pypi.python.org/pypi/bzr</a> best done by running

```
python setup.py register
```

Remember to check the results afterwards.

To be able to register the release you must create an account on <<u>http://pypi.python.org/pypi></u> and have one of the existing owners of the project add you to the group.

## Merging the released code back to trunk

Merge the release branch back into the trunk. Check that changes in NEWS were merged into the right sections. If it's not already done, advance the version number in bzr and bzrlib/\_\_init\_\_.py. Submit this back into pqm for bzr.dev.

As soon as you change the version number in trunk, make sure you have created the corresponding milestone to ensure the continuity in bug targeting or nominating. Depending on the change, you may even have to create a new series (if your change the major or minor release number), in that case go to *Starting a cycle* and follow the instructions from there.

You should also merge (not pull) the release branch into lp:~bzr/bzr/current, so that branch contains the current released code at any time.

## Releases until the final one

Congratulations - you have made your first release. Have a beer or fruit juice - it's on the house! If it was a beta, or candidate, you're not finished yet. Another beta or candidate or hopefully a final release is still to come.

The process is the same as for the first release. Goto Starting the release phase and follow the instructions again. Some details change between beta, candidate and final releases, but they should be documented. If the instructions aren't clear enough, please fix them.

## See also

- Packaging into the bzr PPA to make and publish Ubuntu packages.
- Bazaar Developer Document Catalog
- Development cycles: things that happen during the cycle before the actual release.

Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

In the second se

# Managing the Bazaar PPA

See also: Bazaar Developer Document Catalog.

## Background

We build Ubuntu .deb packages for Bazaar as an important part of the release process. These packages are hosted in a few Personal Package Archives (PPA) on Launchpad.

As of June 2008, there are three PPAs:

<https://launchpad.net/~bzr/+archive> Final released versions.

<https://launchpad.net/~bzr-beta-ppa/+archive> Releases and release candidates.

<https://launchpad.net/~bzr-nightly-ppa/+archive>

Automatic nightly builds from trunk.

We build packages for every supported Ubuntu release <<u>https://wiki.ubuntu.com/Releases</u>>. Packages need no longer be updated when the release passes end-of-life because all users should have upgraded by then. (As of May 2008, Edgy Eft is no longer supported.)

We build a distinct package for each distrorelease. As of bzr 1.5, Dapper uses python-support and later distributions use python-central. If you upload a release-specific version, you should add a suffix to the package version, e.g. bzr.1.3-1~bazaar1~dapper1.

Every package is first uploaded into the beta ppa. For final release versions it is also copied to the main PPA.

The packaging information is kept in branches of bzr on Launchpad, named like <<u>https://code.launchpad.net/~bzr/bzr/packaging-hardy</u>>. or <lp:~bzr/bzr/packaging-hardy>. These branches are intended to be used with the bzr-builddeb plugin.
### Preconditions

- You must have a Launchpad account and be a member of the teams that own these PPAs (~bzr, ~bzr-beta-ppa).
- You must have a GPG key registered to your Launchpad account.
- Configure dput to upload to our PPA with this section in your ~/.dput.cf:

```
[bzr-beta-ppa]
fqdn = ppa.launchpad.net
method = ftp
incoming = ~bzr-beta-ppa/ubuntu
login = anonymous
allow_unsigned_uploads = 0
[bzr-ppa]
fqdn = ppa.launchpad.net
method = ftp
incoming = ~bzr/ubuntu
login = anonymous
allow_unsigned_uploads = 0
```

You may also want to add these lines to prevent inadvertently attempting to upload into Ubuntu or Debian, which will give a somewhat unclear error:

```
[DEFAULT]
default_host_main = notspecified
```

• Configure bzr-builddeb to sign the package, which is required for Launchpad to build it. Put this in ~/.bazaar/builddeb.conf

```
[BUILDDEB]
builder = dpkg-buildpackage -rfakeroot
source-builder= dpkg-buildpackage -rfakeroot -S -sa
```

• You need a Ubuntu (or probably Debian) machine, and

```
sudo apt-get install build-essential devscripts dput quilt
```

Please update this document if you encounter unmet dependencies or find a shorter way to express them.

• You will also want to have the bzr-builddeb plugin installed, which depends on bzrtools.

# **Packaging Bazaar**

#### Short form

For people who have already set up everything they need, building the release packages is as simple as:

```
cd ~/dev/bzr/releases/packaging
export VERSION="1.17~rc1-1~bazaar1"
export PACKAGE="bzr"
export UBUNTU_RELEASES="dapper hardy intrepid jaunty karmic"
~/dev/bzr/bzr.dev/tools/packaging/update-packaging-branches.sh
~/dev/bzr/bzr.dev/tools/packaging/update-changelogs.sh
~/dev/bzr/bzr.dev/tools/packaging/update-control.sh 1.16 1.17 1
~/dev/bzr/bzr.dev/tools/packaging/build-packages.sh
dput bzr-beta-ppa ${PACKAGE}_$VERSION*.changes
```

Rinse and repeat for all the plugins by changing VERSION and PACKAGE.

#### Long Form

 You will end up checking out a separate directory for each supported release. Such as ~/dev/bzr/releases/packaging/hardy. In each of these branches, you will produce the package for the release.

The scripts will also create the branches and produce packages for bzrtools and bzr-svn.

2. Decide on the final version number. It should be of this form:

bzr-1.17~rc1-1~bazaar1~hardy1

Note: There are three hyphen-separated parts: the package

name, the upstream version, and the packaging version.

**Caution:** Upstream betas or release candidates must insert a tilde to make them sort before the final release, like this: bzr-1.17~rc1-1~bazaar1~hardy1.

Final releases will use a release string of the form: bzr-1.17-1~bazaar1~hardy1

Set this base of this up as a usable environment variable:

```
export VERSION="1.17~rc1-1~bazaar1"
```

3. Export the distroreleases that you will be packaging for:

export UBUNTU\_RELEASES="dapper hardy intrepid jaunty karmic

4. Export the program you are packaging:

export PACKAGE="bzr"

5. Checkout (or update) the packaging branch for each supported release:

bzr co lp:~bzr/bzr/packaging-hardy

There is a script available to help:

tools/packaging/update-packaging-branches.sh

- 6. The bzr-buildeb step will download the original tarball if you do not already have it, putting it into a tarballs directory.
- 7. For Bazaar plugins, change the debian/control file to express a dependency on the correct version of bzr.

For bzrtools this is typically:

```
Build-Depends-Indep: bzr (>= 1.17~), rsync
Depends: ${python:Depends}, bzr (>= 1.17~), bzr (<< 1.18~),
</pre>
```

There is a helper script which will update the control file and commit it for all of your \$UBUNTU\_RELEASES. It is available as:

tools/packaging/update-control.sh

You must supply the versions as arguments as follows OLD\_VERSION CURRENT\_VERSION NEXT\_VERSION, such as:

tools/packaging/update-control.sh 1.16 1.17 1.18

8. Make a new debian/changelog entry for the new release, either by using dch or just editing the file:

```
dch -v '1.17~rc1-1~bazaar1~hardy1' -D hardy
```

dch will default to the distro you're working in and this isn't checked against the version number (which is just our convention), so make sure to specify it.

Make sure you have the correct email address for yourself (you may need export DEBEMAIL=`bzr whoami` if it isn't already set), version number, and distribution. It should look something like this:

```
bzr (1.17~rc1-1~bazaar1~hardy1) hardy; urgency=low
 * New upstream release.
 -- John Sample <sample@example.com> Mon, 31 Mar 2008 12:36
```

If you need to upload the package again to fix a problem, normally you should increment the last number in the version number, following the distro name. Make sure not to omit the initial -1, and make sure that the distro name in the version is consistent with the target name outside the parenthesis.

You will also want to commit these changes into the packaging branch.

There is a helper script which will build all the packages for all of your \$UBUNTU\_RELEASES. It is available as:

tools/packaging/update-changelogs.sh

9. Build the source packages:

cd packaging-\$DISTRO; bzr builddeb -S

This will create a .changes file. If you didn't configure builddeb to automatically sign them, you can use

debsign -m\$UID \*.changes

where surd is the gpg key you want to use to sign the changes.

There is a helper script which will build the package for all of your \$UBUNTU\_RELEASES. It is available as:

tools/packaging/build-packages.sh

10. Upload into the beta PPA for each release:

dput bzr-beta-ppa bzr\*1.17-1\*.changes

11. For final release versions, also copy it into the ~bzr PPA:

```
dput bzr-ppa ../bzr_1.17-1\~bazaar1\~hardy1\_source.changes
_______
```

Alternatively, you can use Launchpad's "copy" feature to copy the packages between repositories.

12. You should soon get an "upload accepted" mail from Launchpad, which means that your package is waiting to be built. You can then track its progress in <https://launchpad.net/~bzr-beta-ppa/+archive> and <https://launchpad.net/~bzr-beta-ppa/+archive/+builds>.

#### Packaging bzr-svn

bzr-svn uses a packaging branch that contains both the source (including any changes against upstream) and the debian/ directory.

To build bzr-svn:

- 1. Get a checkout of lp:~bzr/bzr-svn/hardy-ppa/
- 2. Merge from http://bzr.debian.org/pkg-bazaar/bzrsvn/unstable/

This should bring in both upstream and packaging changes for the new release, and it's updated as part of the bzr-svn release process.

It's quite possible you will need to resolve some conflicts.

- 3. Run dch -v 0.4.15-1~bazaar1-hardy1 -D hardy Or similar
- 4. Run bzr builddeb --source

bzr-builddeb will automatically check out the appropriate tag from the main branch of bzr-svn, build, and package it.

### 5. dput bzr-beta-ppa

../bzr-svn\_0.4.15-

1~bazaar1~hardy1\_source.changes

## Monitoring the contents of PPAs

If you add all the bzr PPAs to your sources.list then you can see a summary of current package versions with:

apt-cache madison bzr

Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

In the second se

# Bazaar Windows EC2 Server

We have an Amazon EC2 virtual machine called Desolation for building Windows packages and general testing on Windows. As of 2009-02-19, this is just experimental and this is a draft specification, but we aim to use it for the production Windows installer build of 1.13 in March.

See also:

• Bazaar Developer Documentation Catalog.

# Goals

- The instance is only running (and incurring charges) when it's needed for testing or packaging.
- It can be started or stopped by anyone on the team using a straightforward script.
- Multiple people can get into the same instance at the same time, e.g. if one person needs to pass work on to some one else.
- We keep snapshot of the OS and tool chain so that we can roll back if we need to.
- bzr branches and similar information are kept on stable storage that survives rollbacks of the OS state, and that can be backed up.

Later on we may try automated Windows testing in a similar setup.

# Approach

The working disk and the AMI images are stored in one person's account for billing purposes.

Ideally we want to give other people access to run this machine without giving full access to the account. I'm not sure if that's feasible. If it's not, we might need to allow people to launch the image within their own account; this may be problematic if the shared volume is already in use by someone else.

I don't think it's possible to have an EBS that's shared across accounts, and they can't be attached to multiple running instances. So for now it's probably best to just ignore the concept and store the working data on the instance's local storage, and to copy things up e.g. to Launchpad as required.

On this machine, c: should be used only for the Windows system files, D: for installed programs and working directories, and other drive letters can be used later for mounting EBS storage if desired.

Through ec2-modify-image-attribute we can allow nominated users to access an existing image. We need to have their AWS opaque ID.

Through ec2-bundle-image we can make a new snapshot at any point, which will be stored into the current user's S3 account.

We'll (probably) have one shared account for running builds which is also an administrator for ease of installing software.

You do need to have an RSA keypair to get the initial password for a Windows machine, even though you can't use it to log in later. ec2-get-password takes the full path to the private key to obtain the password from Amazon, and ec2-add-keypair creates a named

keypair at Amazon and returns the private path. One keypair is all that is needed. This is distinct from the account identifier - likely due to the different toolchains in use (the keypairs are used for unix ssh keys, and I (Robert) suspect a rather unix friendly core at Amazon). Once a custom image is made with a saved password, you can skip using ec2-get-password (which is only needed for Windows anyway).

It would be nice if rdesktop could use private key authentication but apparently not.

Should check how the Launchpad ec2test scripts work.

## Procedures

#### Preparation

- Be in the bzr core team. If you are interested in helping with Windows packaging, testing or development just ask.
- Install the Amazon EC2 API tools (needs-packaging bug 330930)
- Create an Amazon Web Services account, sign up for S3 and EC2, and do the various steps to create authentication devices.
- Create a private key and certificate for yourself. Check these environment variables are set and exported, e.g. by setting them in the file ~/.aws. Make sure the files are private.:

```
export EC2_PRIVATE_KEY=~/.ec2/pk-XXXXX.pem
export EC2_CERT=~/.ec2/cert-XXXXX.pem
export EC2_HOME=~/build/ec2-api-tools-1.3-30349
export AWS_SECRET_ACCESS_KEY=XXXXXXXX
export AWS_ACCESS_KEY_ID=XXXXXXXXXX
export EC2_KEYPAIR_NAME=XXXXXXXXX
export EC2_KEYPAIR_NAME=XXXXXXXXX
export PATH=$PATH:$EC2_HOME/bin
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
ssh-add ~/.ec2/id_rsa
```

You can now '. ~/.aws' to get the ec2 commands available.

- (Unix images only) run ec2-add-keypair SOMENAME, e.g. 'bzr'. Put the result (minus the first line) somewhere like ~/.ec2/id\_rsa and chmod go-rw.
- A useful Unix image is ami-bdfe19d4, Eric Hammonds 64-bit Ubuntu image.

- Install the rdesktop client, to actually access the machine.
- Possibly read some of the EC2 documentation for background.
- Create a security group for your that allows rdesktop access and icmp with:

```
ec2-add-group desolation-group -d 'bzr win32 build machine'
ec2-authorize desolation-group -p 3389 -s 1.2.3.4/32
ec2-authorize desolation-group -t -1:-1 -P icmp
```

Add your public IP there. You can repeat that command to allow others in.

#### To start up an instance

- 1. Get the right AMI image ID from another developer.
- 1. Start the instance:

ec2-run-instances \$image\_id -g desolation-group

This will print out some information including the image id, something like i-31a74258.

1. Actually starting the machine will take a few minutes. Once it's in the *running* state, get the machine's public IP with

ec2-describe-instances

1. and then connect

rdesktop -g 1200x850 -u Administrator \$machine\_ip

Don't forget to shut it down when you're done, and check with ec2describe-instances that it did terminate.

#### To save a system snapshot as an image

Bundle the current state. Doing this will reboot the machine. You need to choose a unique s3 bucket name, typically based on a domain or email address, which can contain any number of images. You also need a name unique within the bucket for this image, like desolation-vs2008-20090219. And finally it needs your AWS S3 access key and secret key, which should be set in ~/.aws:

```
ec2-bundle-instance -b ec2.sourcefrog.net \
    -p desolation-vs2008-2009021 \
    -o "$AWS_ACCESS_KEY_ID" \
    -w "$AWS_SECRET_ACCESS_KEY"
```

1. This will take several minutes: You can check progress with

ec2-describe-bundle-tasks

1. Register the files as an image, e.g.:

ec2-register ec2.sourcefrog.net/desolation-vs2008-2009021 This will give you an AMI id for the image.

1. Give access to other team members identified by their Amazon account id:

ec2-modify-image-attributes \$ami\_id -l -a 123412341234

#### Management console (useful!)

https://console.aws.amazon.com/ec2/home

In the second se

In the second se

# Bazaar Architectural Overview

This document describes the key classes and concepts within Bazaar. It is intended to be useful to people working on the Bazaar codebase, or to people writing plugins.

If you have any questions, or if something seems to be incorrect, unclear or missing, please talk to us in irc://irc.freenode.net/#bzr, or write to the Bazaar mailing list. To propose a correction or addition to this document, send a merge request or new text to the mailing list.

The current version of this document is available in the file doc/developers/overview.txt in the source tree, and available online within the developer documentation, <a href="http://doc.bazaar-vcs.org/developers/">http://doc.bazaar-vcs.org/developers/</a>>.

# **Essential Domain Classes**

The core domain objects within the bazaar model are:

- Transport
- Branch
- Repository
- WorkingTree

Transports are explained below. See <a href="http://bazaar-vcs.org/Classes/">http://bazaar-vcs.org/Classes/</a> for an introduction to the other key classes.

## Transport

The Transport layer handles access to local or remote directories. Each Transport object acts as a logical connection to a particular directory, and it allows various operations on files within it. You can *clone* a transport to get a new Transport connected to a subdirectory or parent directory.

Transports are not used for access to the working tree. At present working trees are always local and they are accessed through the regular Python file I/O mechanisms.

#### Filenames vs URLs

Transports work in terms of URLs. Take note that URLs are by definition only ASCII - the decision of how to encode a Unicode string into a URL must be taken at a higher level, typically in the Store. (Note that Stores also escape filenames which cannot be safely stored on all filesystems, but this is a different level.)

The main reason for this is that it's not possible to safely roundtrip a URL into Unicode and then back into the same URL. The URL standard gives a way to represent non-ASCII bytes in ASCII (as %-escapes), but doesn't say how those bytes represent non-ASCII characters. (They're not guaranteed to be UTF-8 – that is common but doesn't happen everywhere.)

For example, if the user enters the URL <a href="http://example/%e0">http://example/%e0</a>, there's no way to tell whether that character represents "latin small letter a with grave" in iso-8859-1, or "latin small letter r with acute" in iso-8859-2, or malformed UTF-8. So we can't convert the URL to Unicode reliably. Equally problematic is if we're given a URL-like string containing (unescaped) non-ASCII characters (such as the accented a). We can't be sure how to convert that to a valid (i.e. ASCII-only) URL, because we don't know what encoding the server expects for those characters. (Although it is not totally reliable, we might still accept these and assume that they should be put into UTF-8.)

A similar edge case is that the URL <a href="http://foo/sweet%2Fsour">http://foo/sweet%2Fsour</a> contains one directory component whose name is "sweet/sour". The escaped slash is not a directory separator, but if we try to convert the URL to a regular Unicode path, this information will be lost.

This implies that Transports must natively deal with URLs. For simplicity they *only* deal with URLs; conversion of other strings to URLs is done elsewhere. Information that Transports return, such as from list\_dir, is also in the form of URL components.

# Repository

Repositories store committed history: file texts, revisions, inventories, and graph relationships between them.

#### Stacked Repositories

A repository can be configured to refer to a list of "fallback" repositories. If a particular revision is not present in the original repository, it refers the query to the fallbacks.

Compression deltas don't span physical repository boundaries. So the first commit to a new, empty repository with fallback repositories will store a full text of the inventory, and of every new file text.

At runtime, repository stacking is actually configured by the branch, not the repository. So doing a\_bzrdir.open\_repository() gets you just the single physical repository, while a\_bzrdir.open\_branch().repository gets one configured with a stacking. Therefore, to permanently change the fallback repository stored on disk, you must use Branch.set\_stacked\_on\_url.

Changing away from an existing stacked-on URL will copy across any necessary history so that the repository remains usable.

A repository opened from an HPSS server is never stacked on the server side, because this could cause complexity or security problems with the server acting as a proxy for the client. Instead, the branch on the server exposes the stacked-on URL and the client can open that.

In the second se

# Integrating with Bazaar

This page should hopefully become a quick guide to integrating other (Python-based) software with Bazaar.

## Manipulating the Working Tree

Most objects in Bazaar are in files, named after the class they contain. To manipulate the Working Tree we need a valid WorkingTree object, which is loaded from the workingtree.py file, eg:

from bzrlib import workingtree
wt = workingtree.WorkingTree.open('/home/jebw/bzrtest')

This gives us a WorkingTree object, which has various methods spread over itself, and its parent classes MutableTree and Tree - its worth having a look through these three files (workingtree.py, mutabletree.py and tree.py) to see which methods are available.

### Compare trees

There are two methods for comparing trees: <a href="changes.from">changes.from</a> and <a href="iter\_changes">iter\_changes</a> is more regular and precise, but it is somewhat harder to work with. See the API documentation for more details.

changes\_from creates a Delta object showing changes:

```
from bzrlib import delta
changes = wt.changes_from(wt.basis_tree())
```

This gives us a Delta object, which has several lists of files for each type of change, eg changes.added is a list of added files, changes.removed is list of removed files, changes.modified is a list of modified files. The contents of the lists aren't just filenames, but include other information as well. To grab just the filename we want the first value, eg:

```
print("list of newly added files")
for filename in changes.added:
    print("%s has been added" % filename[0])
```

The exception to this is changes.renamed, where the list returned for each renamed files contains both the old and new names – one or both may interest you, depending on what you're doing.

For example:

```
print("list of renamed files")
for filename in changes.renamed:
    print("%s has been renamed to %s" % (filename[0], filename[1]
```

# Adding Files

If you want to add files the same way bzr add does, you can use MutableTree.smart\_add. By default, this is recursive. Paths can either be absolute or relative to the workingtree:

For more precise control over which files to add, use MutableTree.add:

```
wt.add(['dir1/filea.txt', 'fileb.txt', '/home/jebw/bzrtesttree/
]
```

# **Removing Files**

You can remove multiple files at once. The file paths need to be relative to the workingtree:

```
wt.remove(['filea.txt', 'fileb.txt', 'dir1'])
```

By default, the files are not deleted, just removed from the inventory. To delete them from the filesystem as well:

```
wt.remove(['filea.txt', 'fileb.txt', 'dir1'], keep_files=False)
```

# Renaming a File

You can rename one file to a different name using WorkingTree.rename\_one. You just provide the old and new names, eg:

wt.rename\_one('oldfile.txt','newfile.txt')

# Moving Files

You can move multiple files from one directory into another using WorkingTree.move:

```
wt.move(['olddir/file.txt'], 'newdir')
```

More complicated renames/moves can be done with transform.TreeTransform, which is outside the scope of this document.

# **Committing Changes**

To commit \_all\_ the changes to our working tree we can just call the WorkingTree's commit method, giving it a commit message, eg:

wt.commit('this is my commit message')

To commit only certain files, we need to provide a list of filenames which we want committing, eg:

## Generating a Log for a File

Generating a log is, in itself, simple. Grab a branch (see below) and pass it to show\_log together with a log formatter, eg:

```
from bzrlib import log
from bzrlib import branch
b = branch.Branch.open('/path/to/bazaar/branch')
lf = log.LongLogFormatter(to_file=sys.stdout)
log.show_log(b, lf)
```

Three log formatters are included with bzrlib: LongLogFormatter, ShortLogFormatter and LineLogFormatter. These provide long, short and single-line log output formats. It's also possible to write your own in very little code.

## Annotating a File

To annotate a file, we want to walk every line of a file, retrieving the revision which last modified/created that line and then retrieving the information for that revision.

First we get an annotation iterator for the file we are interested in:

```
tree, relpath = workingtree.WorkingTree.open_containing('/path/
fileid = tree.path2id(relpath)
annotation = list(tree.annotate_iter(fileid))
```

To avoid repeatedly retrieving the same revisions we grab all revisions associated with the file at once and build up a map of id to revision information. We also build an map of revision numbers, again indexed by the revision id:

```
revision_ids = set(revision_id for revision_id, text in annotat
revisions = tree.branch.repository.get_revisions(revision_ids)
revision_map = dict(izip(revision_ids, revisions))
revno_map = tree.branch.get_revision_id_to_revno_map()
```

Finally, we use our annotation iterator to walk the lines of the file, displaying the information from our revision maps as we go:

```
for revision_id, text in annotation :
    rev = revision_map[revision_id]
    revno = revno_map[revision_id]
    revno_string = '.'.join(str(i) for i in revno)
    print "%s, %s: %s" % (revno_string, rev.committer, text)
```
# Working with branches

To work with a branch you need a branch object, created from your branch:

```
from bzrlib import branch
```

```
b = branch.Branch.open('/home/jebw/bzrtest')
```

### Branching from an existing branch

To branch you create a branch object representing the branch you are branching from, and supply a path/url to the new branch location. The following code clones the bzr.dev branch (the latest copy of the Bazaar source code) - be warned it has to download 60meg so takes a while to run with no feedback:

```
from bzrlib import branch
b = branch.Branch.open('http://bazaar.launchpad.net/~bzr-pqm/bz
nb = b.bzrdir.sprout('/tmp/newBzrBranch').open_branch()
```

This provides no feedback, since Bazaar automatically uses the 'silent' UI.

### Pushing and pulling branches

To push a branch you need to open the source and destination branches, then just call push with the other branch as a parameter:

```
from bzrlib import branch
b1 = branch.Branch.open('file:///home/user/mybranch')
b2 = branch.Branch.open('http://bazaar.launchpad.net/~bzr-pqm/b
b1.push(b2)
```

Pulling is much the same:

b1.pull(b2)

If you have a working tree, as well as a branch, you should use WorkingTree.pull, not Branch.pull.

This won't handle conflicts automatically though, so any conflicts will be left in the working tree for the user to resolve.

### Checkout from an existing branch

This performs a Lightweight checkout from an existing Branch:

```
from bzrlib import bzrdir
accelerator_tree, source = bzrdir.BzrDir.open_tree_or_branch('h
source.create_checkout('/tmp/newBzrCheckout', None, True, accel
]
```

To make a heavyweight checkout, change the last line to:

```
source.create_checkout('/tmp/newBzrCheckout', None, False, acce
```

### **History Operations**

### Finding the last revision number or id

To get the last revision number and id of a branch use:

revision\_number, revision\_id = branch.last\_revision\_info()

If all you care about is the revision\_id there is also the method:

revision\_id = branch.last\_revision()

# Getting the list of revision ids that make up a branch

IMPORTANT: This should be avoided wherever possible, as it scales with the length of history:

```
revisions = branch.revision_history()
```

now revisions[0] is the revision id of the first commit, and revs[-1] is the revision id of the most recent. Note that if all you want is the last revision then you should use branch.last\_revision() as described above, as it is vastly more efficient.

### Getting a Revision object from a revision id

The Revision object has attributes like "message" to get the information about the revision:

```
repo = branch.repository
revision = repo.get_revision(rev_id)
```

#### Accessing the files from a revision

To get the file contents and tree shape for a specific revision you need a RevisionTree. These are supplied by the repository for a specific revision id:

revtree = repo.revision\_tree(rev\_id)

RevisionTrees, like all trees, can be compared as described in "Comparing Trees" above.

The most common way to list files in a tree is Tree.iter\_entries(). The simplest way to get file content is Tree.get\_file(). The best way to retrieve file content for large numbers of files *Tree.iter\_files\_bytes()*`

Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

In the second se

# **Bazaar Design Principles**

We have learned or adopted a few general principles for code in Bazaar. Generally we will try to follow them in future, either for consistency or because they've been proven to work well, or both.

We may need to depart from these principles in particular special cases, or modify them as we learn more, or we might be diverging for them for no very good reason but just because of bugs. If in doubt, ask.

See also: Bazaar Developer Document Catalog.

## Testing

Untested code is broken code.

So if a function is removed from the normal flow of execution (perhaps because a new default format was introduced) we have to make sure we can still execute and test the old code – or remove it altogether.

### Data formats

Fixing code once it's released is easy; fixing a problematic data format once people have started using it is more difficult. We should document and review formats separately from the code that implements them.

Data formats should have clear format markers that allow us to support new formats in future. It should be easy to read the format without reading the whole object.

The format marker should be a string understandable by a user that names the format and gives the bzr release that introduced it. If the bzr program doesn't understand that format, it can at least show that format marker to the user.

Once we mark a format as supported, we'll continue supporting it for several future releases, and support upgrading from it forever.

Once we've released a format, we normally don't change it. Adding new optional elements can cause problems when older clients don't understand those changes, or don't propagate them properly.

We clearly distinguish internal files from user files. Files inside .bzr/ are only written to by bzr and we discourage users from editing them. Within bzr, code addressing the abstract interface of the Branch, BzrDir, etc shouldn't know where or how the internal files are stored. If anything else is written in there, it won't be propagated when pushing or pulling, and won't be converted when upgrading. (This is not quite true though; there is a branch.conf.)

User files within the tree, by contrast, we always store and return verbatim. It's OK for Bazaar to read and act on these files (as we do with .bzrignore), and to update them (as bzr ignore does), but they

remain clearly user files and can be directly edited.

♦ Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

In the second se

# Plans

- Performance roadmap The roadmap for fixing performance in bzr over the next few releases.
- Co-located branches Planned(?) support for storing multiple branches in one file-system directory.
- Bazaar Windows Shell Extension Options Implmentation strategy for Bazaar Windows Shell Extensions, aka TortoiseBzr.
- CHK Optimized index

igoplus Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

Whome | Documentation | Developer Document Catalog (2.2b1) » Plans » previous | next

# 1 Bazaar Performance Roadmap

#### Contents

- 1 Bazaar Performance Roadmap
  - 1.1 About the performance roadmap
    - 1.1.1 What should be in the roadmap?
    - 1.1.2 What should the final system look like, how is it different to what we have today?
    - 1.1.3 What use cases should be covered?
    - 1.1.4 How is development on the roadmap coordinated?
    - 1.1.5 Planned changes to the bzr core
      - 1.1.5.1 Library changes
      - 1.1.5.2 Interoperable disk changes
      - 1.1.5.3 Possibly non-interoperable disk changes
      - 1.1.5.4 Non-interoperable disk changes
    - 1.1.6 Integration of performance changes
  - 1.2 Analysis of use cases
    - 1.2.1 Analysing a specific use case
    - 1.2.2 Performing the analysis
    - 1.2.3 What factors should be considered?

#### • 1.3 Use cases

- 1.3.1 Initial push / pull
  - 1.3.1.1 Optimal case
  - 1.3.1.2 Disk case
  - 1.3.1.3 Smart Network Case
    - 1.3.1.3.1 Phase 1
    - 1.3.1.3.2 Phase 2
  - 1.3.1.4 Dumb Network Case
  - 1.3.1.5 Wants
- 1.3.2 Incremental push/pull

- 1.3.2.1 Functional Requirements
- 1.3.2.2 Factors which should add work for push/pull
- 1.3.2.3 Push/pull overview
  - 1.3.2.3.1 New data identification
    - 1.3.2.3.1.1 Set synchronisation approaches
    - 1.3.2.3.1.2 DAG synchronisation approaches
    - 1.3.2.3.1.3 File level scaling
    - 1.3.2.3.1.4 API scaling
  - 1.3.2.3.2 Data reading
    - 1.3.2.3.2.1 File level scaling
    - 1.3.2.3.2.2 API scaling
  - 1.3.2.3.3 Data Verification and writing
    - 1.3.2.3.3.1 Overview summary
    - 1.3.2.3.3.2 File level scaling
    - 1.3.2.3.3.3 API scaling
- 1.3.2.4 Notes from London
- 1.3.3 Add
  - 1.3.3.1 Least work we can hope to perform
  - 1.3.3.2 Per file algorithm
- 1.3.4 Commit Performance Notes
  - 1.3.4.1 Changes to commit
  - 1.3.4.2 Commit: The Minimum Work Required
  - 1.3.4.3 Commit vs Status
  - 1.3.4.4 Avoiding Work: Smarter Change Detection
  - 1.3.4.5 Avoiding Work: Better Layering
  - 1.3.4.6 Avoiding work: avoiding reading parent data
  - 1.3.4.7 Code structure
  - 1.3.4.8 Complications of commit
  - 1.3.4.9 Interface stack

- 1.3.4.10 Branch->Tree interface
- 1.3.4.11 Information from the tree to repository
- 1.3.4.12 Information from the repository to the tree
- 1.3.4.13 Selective commit
- 1.3.4.14 Common commit code
- 1.3.4.15 Order of traversal
- 1.3.4.16 Open question: per-file graphs
- 1.3.5 diff Performance Analysis
  - 1.3.5.1 Minimal Work
    - 1.3.5.1.1 Reuse of historical comparisons
    - 1.3.5.1.2 Historical Tree Against Historical Tree
    - 1.3.5.1.3 Basis Against Historical Tree
    - 1.3.5.1.4 Basis Against Basis
    - 1.3.5.1.5 Working Tree Against Basis
    - 1.3.5.1.6 Working Tree Against Historical Tree
    - 1.3.5.1.7 Working Tree Against Working Tree
  - 1.3.5.2 API Changes
  - 1.3.5.3 Storage considerations
- 1.3.6 Garbage Collection
  - 1.3.6.1 Least work we can hope to perform
- 1.3.7 Revert
  - 1.3.7.1 Least work we can hope to perform
- 1.3.8 The status command
  - 1.3.8.1 UI Overview
  - 1.3.8.2 Ideal work for working tree to historical status
  - 1.3.8.3 Locality of reference
  - 1.3.8.4 Scaling observations
- 1.3.9 Annotate
- 1.3.10 Scaling analysys of Merge

- 1.3.10.1 Needs
- 1.3.10.2 Notes
- 1.3.11 Bundle Creation
  - 1.3.11.1 Needs
- 1.3.12 Uncommit Performance Notes
  - 1.3.12.1 Specification of uncommit
- 1.3.13 Missing
- 1.4 Subsystem designs
  - 1.4.1 Directory fingerprints
    - 1.4.1.1 Introduction
    - 1.4.1.2 Use-case oriented APIs
      - 1.4.1.2.1 commit
      - 1.4.1.2.2 log
    - 1.4.1.3 Open questions
    - 1.4.1.4 Conclusions
    - 1.4.1.5 Design changes
    - 1.4.1.6 API changes

## 1.1 About the performance roadmap

### 1.1.1 What should be in the roadmap?

A good roadmap provides a place for contributors to look for tasks, it provides users with a sense of when we will fix things that are affecting them, and it also allows us all to agree about where we are headed. So the roadmap should contain enough things to let all this happen.

I think that it needs to contain the analysis work which is required, a list of the use cases to be optimised, the disk changes required, and the broad sense of the api changes required. It also needs to list the inter-dependencies between these things: we should aim for a large surface area of 'ready to be worked on' items, that makes it easy to improve performance without having to work in lockstep with other developers.

Clearly the analysis step is an immediate bottleneck - we cannot tell if an optimisation for use case A is a pessimism for use case B until we have analysed both A and B. I propose that we complete the analysis of say a dozen core use cases end to end during the upcoming sprint in London. We should then be able to fork() for much of the detailed design work and regroup with disk and api changes shortly thereafter.

I suspect that clarity of layering will make a big difference to developer parallelism, so another proposal I have is for us to look at the APIs for Branch and Repository in London in the light of what we have learnt over the last years.

1.1.2 What should the final system look like, how is it different to what we have today?

One of the things I like the most about bzr is its rich library API, and I've heard this from numerous other folk. So anything that will remove that should be considered a last resort.

Similarly our relatively excellent cross platform support is critical for projects that are themselves cross platform, and thats a considerable number these days.

And of course, our focus on doing the right thing is what differentiates us from some of the other VCS's, so we should be focusing on doing the right thing quickly :).

What we have today though has grown organically in response to us identifying bottlenecks over several iterations of back end storage, branch metadata and the local tree representation. I think we are largely past that and able to describe the ideal characteristics of the major actors in the system - primarily Tree, Branch, Repository - based on what we have learnt.

#### 1.1.3 What use cases should be covered?

My list of use cases is probably not complete - its just the ones I happen to see a lot :). I think each should be analysed comprehensively so we dont need to say 'push over the network' - its implied in the scaling analysis that both semantic and file operation latency will be considered.

These use cases are ordered by roughly the ease of benchmarking, and the frequency of use. This ordering is so that when people are comparing bzr they are going to get use cases we have optimised; and so that as we speed things up our existing users will have the things they do the most optimised.

- status tree
- status subtree

- commit
- commit to a bound branch
- incremental push/pull
- log
- log path
- add
- initial push or pull [both to a new repo and an existing repo with different data in it]
- diff tree
- diff subtree
- revert tree
- revert subtree
- merge from a branch
- merge from a bundle
- annotate
- create a bundle against a branch
- uncommit
- missing
- update
- cbranch

# 1.1.4 How is development on the roadmap coordinated?

I think we should hold regular get-togethers (on IRC) to coordinate on our progress, because this is a big task and its a lot easier to start helping out some area which is having trouble if we have kept in contact about each areas progress. This might be weekly or fortnightly or some such.

we need a shared space to record the results of the analysis and the roadmap as we go forward. Given that we'll need to update these as new features are considered, I propose that we use doc/design as a working space, and as we analyse use cases we include them in there - including the normal review process for each patch. We also need documentation about doing performance tuning - not the minutiae, though that is needed, but about how to effective choose things to optimise which will give the best return on time spent - that is what the roadmap should help with, but this looks to be a large project and an overview will be of great assistance I think. We want to help everyone that wishes to contribute to performance to do so effectively.

Finally, its important to note that coding is not the only contribution testing, giving feedback on current performance, helping with the analysis are all extremely important tasks too and we probably want to have clear markers of where that should be done to encourage such contributions.

#### 1.1.5 Planned changes to the bzr core

Delivering the best possible performance requires changing the bzr core design from that present in 0.16. Some of these changes are incremental and can be done with no impact on disk format. Many of them however do require changes to the disk format, and these can be broken into two sets of changes, those which are sufficiently close to the model bzr uses today to interoperate with the 0.16 disk formats, and those that are not able to interoperate with the 0.16 disk formats - specifically some planned changes may result in data which cannot be exported to bzr 0.16's disk formats and then imported back to the new format without losing critical information. If/when this takes place it will be essentially a migration for users to switch from their bzr 0.16 repository to a bzr that supports them. We plan to batch all such changes into one large 'experimental' repository format, which will be complete stable and usable before we migrate it to become a supported format. Getting new versions of bzr in widespread use at that time will be very important, otherwise the user base may be split in two - users that have upgraded and

users that have not.

The following changes are grouped according to their compatability impact: library only, disk format but interoperable, disk format interoperability unknown, and disk format, not interoperable.

#### 1.1.5.1 Library changes

These changes will change bzrlib's API but will not affect the disk format and thus do not pose a significant migration issue.

- For our 20 core use cases, we plan to add targeted API's to bzrlib that are repository-representation agnostic. These will instead reflect the shape of data access most optimal for that case.
- Deprecate 'versioned files' as a library concept. Instead of asking for information about a file-over-time as a special case, we will move to an API that assumes less coupling between the historical information and the ability to obtain texts/deltas etc. Specifically, we need to remove all API's that act in terms of on disk representation except those within a given repository implementation.
- Create a validator for revisions that is more amenable to use by other parts of the code base than just the gpg signing facility. This can be done today without changing disk, possibly with a performance hit until the disk formats match the validatory logic. It will be hard to tell if we have the right routine for that until all the disk changes are complete, so while this is a library only change, its likely one that will be delayed to near the end of the process.
- Add an explicit API for managing cached annotations. While annotations are considered a cache this is not exposed in such a way that cache operations like 'drop the cache' can be performed. On current disk formats the cache is mandatory, but an API to manage would allow refreshing of

the cache (e.g. after ghosts are filled in during baz conversions).

- Use the \_iter\_changes API to perform merges. This is a small change that may remove the need to use inventories in merge, making a dramatic difference to merge performance once the tree shape comparison optimisations are implemented.
- Create a network-efficient revision graph API. This is the logic at the start of push and pull operations, which currently scales O(graph size). Fixing the scaling can be done, but there are tradeoffs to latency and performance to consider, making it a little tricky to get right.
- Working tree disk operation ordering. We plan to change the order in which some operations are done (specifically TreeTransform ones) to improve performance. There is already a 66% performance boost in that area going through review.
- Stop requiring full memory copies of files. Currently bzr requires that it can hold 3 copies of any file its versioning in memory. Solving this is tricky, particularly without performance regressions on small files, but without solving it versioning of .iso and other large objects will continue to be extremely painful.
- Add an API for per-file graph access that allows incremental access and is suitable for on-demand generation if desired.
- Repository stacking API. Allowing multiple databases to be stacked to give a single 'repository' will allow implementation of some long desired features like history horizons, and bundle usage where the bundle is not added to the local repository just to examine its contents.
- Revision data manipulation API. We need a single streaming API for adding data to or getting it from a repository. This will need to allow hints such as 'optimise for

size', or 'optimise for fast-addition' to meet the various users planned, but it is a core part of the library today, and its not sufficiently clean to let us simplify/remove a lot of related code today.

#### 1.1.5.2 Interoperable disk changes

- New container format to allow single-file description of multiple named objects. This will provide the basis for transmission of revisions over the network, the new bundle format, and possibly a new repository format as well. [Core implemented]
- Separate the annotation cache from the storage of actual file texts and make the annotation style, and when to do it, configurable. This will reduce data sent over the wire when repositories have had 'needs-annotations' turned off, which very large trees may choose to do generating just-in-time annotations may be desirable for those trees (even when performing annotation based merges).
- Repository disk operation ordering. The order that tasks access data within the repository and the layout of the data should be harmonised. This will require disk format changes but does not inherently alter the model, so its straight forward to export from a repository that has been optimised in this way to a 0.16 based repository.
- Inventory representation. An inventory is a logical description of the shape of a version controlled tree. Currently we operate on the whole inventory as a tree broken down per directory, but we store it as a flat file. This scale very poorly as even a minor change between inventories requires us to scan the entire file, and in large trees this is many megabytes of data to consider. We are investigating the exact form, but the intent is to change the serialisation of inventories so that comparing two

inventories can be done in some smaller time - e.g. O(log N) scaling. Whatever form this takes, a repository that can export it directly will be able to perform operations between two historical trees much more efficiently than the current repositories.

- Delta storage optimisation. We plan to change the delta storage logic to use a binary delta like xdelta rather than using line based deltas from python. These binary deltas could be done along ancestry ordering, or other arbitrary patterns chosen for their intended use. Line based deltas will still be created for cached annotations. This is still under some discussion. http://bazaarvcs.org/PerformanceRoadmap/Xdelta
- Greatest distance from origin cache. This is a possible change to introduce, but it may be unnecessary listed here for completeness till it has been established as [un]needed.

#### 1.1.5.3 Possibly non-interoperable disk changes

- Removing of derivable data from the core of bzr. Much of the data that bzr stores is derivable from the users source files. For instance the annotations that record who introduced a line. Given the full history for a repository we can recreate that at any time. We want to remove the dependence of the core of bzr on any data that is derivable, because doing this will give us the freedom to:
  - Improve the derivation algorithm over time.
  - Deal with bugs in the derivation algorithms without having 'corrupt repositories' or such things.

However, some of the data that is technically derived, like the per-file merge graph, is both considered core, and can be generated differently when certain circumstances arive, by bzr 0.16. Any change to the 'core' status of that data will discard data that cannot be recreated and thus lead to the inability to export from a format where that is derived data to bzr 0.16's formats without errors occuring in those circumstances. Some of the data that may be considered for this includes:

- Per file merge graphs
- Annotations

#### 1.1.5.4 Non-interoperable disk changes

- Drop the per-file merge graph 'cache' currently held in the FILE-ID.kndx files. A specific case of removing derivable data, this may allow smaller inventory metadata and also make it easier to allow two different trees (in terms of last-change made, e.g. if one is a working tree) to be compared using a hash-tree style approach.
- Use hash based names for some objects in the bzr database. Because it would force total-knowledge-of-history on the graph revision objects will not be namable via hash's and neither will revisio signatures. Other than that though we can in principle use hash's e.g. SHA1 for everything else. There are many unanswered questions about hash based naming related to locality of reference impacts, which need to be answered before this becomes a definite item.

#### 1.1.6 Integration of performance changes

To deliver a version of bzr with all our planned changes will require significant integration work. Minimally each change needs to integrate with some aspect of the bzr version it's merged into, but in reality many of these changes while conceptually independent will in fact have to integrate with the other changes we have planned before can have a completed system. Additionally changes that alter disk formats are inherently more tricky to integrate because we will often need to alter apis throughout the code base to expose the increased or reduced model of the preferred disk format.

You can generate a graph performance.png in the source tree from Graphviz "dot" file performance.dot. This graphs out the dependencies to let us make accurate assessments of the changes needed in terms of code and API, hopefully minimising the number of different integration steps we have to take, while giving us a broad surface area for development. It's based on a summary in the next section of this document of the planned changes with their expected collaborators and dependencies. Where a command is listed, the expectation is that all uses of that command - local, remote, dumb transport and smart transport are being addressed together.

The following provides a summary of the planned changes and their expected collaborators within the code base, along with an estimate of whether they are likely to require changes to their collaborators to be considered 'finished'.

- Use case target APIs: Each of these is likely to alter the Tree interface. Some few of them focus on Branch and will alter Branch and Repository accordingly. As they are targeted APIs we can deep changes all the way down the stack to the underlying representation to make it all fit well. Presenting a top level API for many things will be possible now as long as the exposed data is audited for things we plan to make optional, or remove: Such things cannot be present in the final API. Writing these APIs now will provide strong feedback to the design process for those things which are considered optional or removable, so these APIs should be implemented before removing or making optional existing data.
- Deprecating versioned files as a supported API: This

collaborates with the Repository API but can probably be done by adding a replacement API for places where the versioned-file api is used. We may well want to keep a concept of 'a file over time' or 'inventories over time', so the existing repository model of exposing versioned file objects may be ok; what we need to ensure we do is remove the places in the code base where you create or remove or otherwise describe manipulation of the storage by knit rather than talking at the level of file ids and revision ids. The current versioned-file API would be a burden for implementors of a blob based repository format, so the removal of callers, and deprecation of those parts of the API should be done before creating a blob based repository format.

- Creating a revision validator: Revision validators may depend on storage layer changes to inventories so while we can create a revision validator API, we cannot create the final one until we have the inventory structural changes completed.
- Annotation caching API: This API is a prerequisite for new repository formats. If written after they are introduced we may find that the repository is lacking in functionality, so the API should be implemented first.
- \_iter\_changes based merging: If the current \_iter\_changes\_ API is insufficient, we should know about that before designing the disk format for generating fast \_iter\_changes\_ output.
- Network-efficient revision graph API: This influences what questions we will want to ask a local repository very quickly; as such it's a driver for the new repository format and should be in place first if possible. Its probably not sufficiently different to local operations to make this a hard ordering though.
- Working tree disk ordering: Knowing the expected order for

disk operations may influence the needed use case specific APIs, so having a solid understanding of what is optimal - and why - and whether it is pessimal on non linux platforms is rather important.

- Be able to version files greater than memory in size: This cannot be achieved until all parts of the library which deal with user files are able to provide access to files larger than memory. Many strategies can be considered for this such as temporary files on disk, memory mapping etc. We should have enough of a design laid out that developers of repository and tree logic are able to start exposing apis, and considering requirements related to them, to let this happen.
- Per-file graph access API: This should be implemented on top of or as part of the newer API for accessing data about a file over time. It can be a separate step easily; but as it's in the same area of the library should not be done in parallel.
- Repository stacking API: The key dependency/change required for this is that repositories must individually be happy with having partial data e.g. many ghosts. However the way the API needs to be used should be driven from the command layer in, because its unclear at the moment what will work best.
- Revision stream API: This API will become clear as we streamline commands. On the data insertion side commit will want to generate new data. The commands pull, bundle, merge, push, possibly uncommit will want to copy existing data in a streaming fashion.
- New container format: Its hard to tell what the right way to structure the layering is. Probably having smooth layering down to the point that code wants to operate on the containers directly will make this more clear. As bundles will become a read-only branch & repository, the smart server

wants streaming-containers, and we are planning a pack based repository, it appears that we will have three different direct container users. However, the bundle user may in fact be fake - because it really is a repository.

- Separation of annotation cache: Making the disk changes to achieve this depends on the new API being created. Bundles probably want to be annotation-free, so they are a form of implementation of this and will need the on-demand annotation facility.
- Repository operation disk ordering: Dramatically changing the ordering of disk operations requires a new repository format. We have most of the analysis done to be able to specify the desired ordering, so it should be possible to write such a format now based on the container logic, but without any of the inventory representation or delta representation changes. This would for instance involve pack combining ordering the existing diffs in reverse order.
- Inventory representation: This has a dependency on what data is dropped from the core and what is kept. Without those changes being known we can implement a new representation, but it won't be a final one. One of the services the new inventory representation is expected to deliver is one of validators for subtrees – a means of comparing just subtrees of two inventories without comparing all the data within that subtree.
- Delta storage optimisation: This has a strict dependency on a new repository format. Optimisation takes many forms we probably cannot complete the desired optimisations under knits though we could use xdelta within a knitvariation.
- Greatest distance from origin cache: The potential users of this exist today, it is likely able to be implemented immediately, but we are not sure that its needed anymore, so it is being shelved.

 Removing derivable data: Its very hard to do this while the derived data is exposed in API's but not used by commands. Implemented the targeted API's for our core use cases should allow use to remove accidental use of derived data, making only explicit uses of it visible, and isolating the impact of removing it : allowing us to experiment sensibly. This covers both dropping the per-file merge graph and the hash-based-names proposals.

## 1.2 Analysis of use cases

#### 1.2.1 Analysing a specific use case

The analysis of a use case needs to provide as outputs:

- The functional requirements that the use case has to satisfy.
- The file level operations and access patterns that will give the best performance.
- A low friction API which will allow the use case to be implemented.
- The release of bzr (and thus the supported features) for which the analysis was performed. The feature set of bzr defines the access patterns and data required to implement any use case. So when we add features, their design changes the requirements for the parts of the system they alter, so we need to re-analyse use cases when bzr's feature set changes. If future plans are considered in the analysis with the intention of avoiding rework, these should also be mentioned.

#### 1.2.2 Performing the analysis

The analysis needs to be able to define the characteristics of the involved disk storage and APIs. That means we need to examine the data required for the operation, in what order it is required, on both the read and write sides, and how that needs to be presented to be consistent with our layering.

As a quick example: 'annotation of a file requires the file id looked up from the tree, the basis revision id from the tree, and then the text of that fileid-revisionid pair along with the creating revision id allocated to each line, and the dotted revision number of each of those revision ids.' All three of our key domain objects are involved here, but we haven't defined any characteristics of the api or disk facilities yet. We could then do that by saying something like 'the file-id lookup should degrade gracefully as trees become huge. The tree basis id should be constant time. Retrieval of the annotated text should be roughly constant for any text of the same size regardless of the number of revisions contributing to its content. Mapping of the revision ids to dotted revnos could be done as the text is retrieved, but its completely fine to post-process the annotated text to obtain dotted-revnos.'

### 1.2.3 What factors should be considered?

Obviously, those that will make for an extremely fast system :). There are many possible factors, but the ones I think are most interesting to design with are:

- baseline overhead:
  - The time to get bzr ready to begin the use case.
- scaling: how does performance change when any of the follow aspects of the system are ratcheted massively up or down:
  - number of files/dirs/symlinks/subtrees in a tree (both working and revision trees)
  - size of any particular file
  - number of elements within a single directory
  - length of symlinks
  - number of changes to any file over time (subordinately also the number of merges of the file)
  - number of commits in the ancestry of a branch (subordinately also the number of merges)
  - number of revisions in a repository
  - number of fileids in a repository
  - number of ghosts in a given graph (revision or per-file)

- number of branches in a repository
- number of concurrent readers for a tree/branch/repository
- number of concurrent writers for objects that support that.
- latency to perform file operations (e.g. slow disks, network file systems, our VFS layer and FTP/SFTP/etc)
- bandwidth to the disk storage
- latency to perform semantic operations (hpss specific)
- bandwidth when performing semantic operations.
- locality of reference: If an operation requires data that is located within a small region at any point, we often get better performance than with an implementation of the same operation that requires the same amount of data but with a lower locality of reference. Its fairly tricky to add locality of reference after the fact, so I think its worth considering up front.

Using these factors, to the annotate example we can add that its reasonable to do two 'semantic' round trips to the local tree, one to the branch object, and two to the repository. In file-operation level measurements, in an ideal world there would be no more than one round trip for each semantic operation. What there must not be is one round trip per revision involved in the revisionid->dotted number mapping, nor per each revision id attributed to a line in the text.

Not all the items mentioned above are created equal. The analysis should include the parameters considered and the common case values for each - the optimisation should be around the common cases not around the exceptions.

For instance, we have a smart server now; file level operations are relatively low latency and we should use that as the common case. At this point we intend to preserve the performance of the dumb protocol networking, but focus on improving network performance via the smart server and thus escape the file-level operation latency considerations.

Many performance problems only become visible when changing the scaling knobs upwards to large trees. On small trees its our baseline performance that drives incremental improvements; on large trees its the amount of processing per item that drives performance. A significant goal therefore is to keep the amount of data to be processed under control. Ideally we can scale in a sublinear fashion for all operations, but we MUST NOT scale even linearly for operations that invoke a latency multiplier. For example, reading a file on disk requires finding the inode for the file, then the block with the data and returning the contents. Due to directory grouping logic we pay a massive price to read files if we do not group the reads of files within the same directory.
# 1.3 Use cases

# 1.3.1 Initial push / pull

## 1.3.1.1 Optimal case

(a motivating example of ultimate performance) Assume there is a file with exactly the right data in compressed form. This may be a tarred branch, a bundle, or a blob format. Performance in this case scales with the size of the file.

#### 1.3.1.2 Disk case

Assume current repo format. Attempt to achieve parity with cp -r. Read each file only 1 time.

- read knit graph for revisions
- write filtered copy of revision knit O(d+a)
- write filtered copy of knit index O(d)
- Open knit index for inventory
- Write a filtered copy of inventory knit and simultaneously not all referenced file-ids O(b+d)
- Write filtered copy of inventory knit index O(d)
- For each referenced file-id:
  - Open knit index for each file knit O(e)
  - If acceptable threshold of irrelevant data hard-link O(f)
  - Otherwise write filtered copy of text knit and simultaneously write the fulltext to tree transform O(h)
- Write format markers O(1)
- a: size of aggregate revision metadata
- b: size of inventory changes for all revisions
- c: size of text changes for all files and all revisions (e \* g)

- d: number of relevant revisions
- e: number of relevant versioned files
- **f:** size of the particular versioned file knit index
- **g:** size of the filtered versioned file knit
- **h:** size of the versioned file fulltext
- i: size of the largest file fulltext

## 1.3.1.3 Smart Network Case

## 1.3.1.3.1 Phase 1

Push: ask if there is a repository, and if not, what formats are okay Pull: Nothing

#### 1.3.1.3.2 Phase 2

Push: send initial push command, streaming data in acceptable format, following disk case strategy Pull: receive initial pull command, specifying format

Pull client complexity: O(a), memory cost O(1) Push client complexity: processing and memory cost same as disk case

#### 1.3.1.4 Dumb Network Case

Pull: same as disk case, but request all file knit indices at once and request al file knits at once. Push: same as disk case, but write all files at once.

#### 1.3.1.5 Wants

- Read partial graph
- Read multiple segments of multiple files on http and sftp
- Write multiple files over SFTP

# 1.3.2 Incremental push/pull

This use case covers pulling in or pushing out some number of revisions which is typically a small fraction of the number already present in the target repository. Pushing and pulling are defined as branch level operations for ease of interaction with VCS systems that have no repository abstraction (such as bzr-svn or GNU Arch) but within bzrlib's core they are currently the responsibility of the Repository object.

#### 1.3.2.1 Functional Requirements

A push or pull operation must:

- Copy all the data to reconstruct the selected revisions in the target branch. This is the goal of push and pull after all.
- Reject corrupt data. As bzr has no innate mechanism for discarding corrupted data, corrupted data should not be incorporated accidentally.

#### 1.3.2.2 Factors which should add work for push/pull

- Baseline overhead: The time to connect to both branches.
- Actual new data in the revisions being pulled (drives the amount of data to move around, includes the commit messages etc)
- Number of revisions in the two repositories (scaling affects the determination of what revisions to move around).

#### 1.3.2.3 Push/pull overview

- 1. New data is identified in the source repository.
- 2. That data is read from the source repository.
- 3. The same data is verified and written to the target repository in such a manner that its not visible to readers until its ready for

use.

#### 1.3.2.3.1 New data identification

We have a single top level data object: revisions. Everything else is subordinate to revisions, so determining the revisions to propagate should be all thats needed. This depends on revisions with partial data - such as those with no signature - being flagged in some efficient manner.

We could do this in two manners: determine revisions to sync and signatures to sync in two passes, or change the 'value' of a revision implicitly when the signature is different. E.g. by using merkle hash trees with the signature data a separate component the signatures will naturally be identified to sync.

We want to only exchange data proportional to the number of new revisions and signatures in the system though. One way to achieve this for revisions is to walk the graph out from the desired tips until the surface area intersection is found. For signatures a set difference seems to be needed as there is no DAG of signatures: the presence of one has no implications on the presence of another, so a full pass over the set of signatures would be required to confirm no new signatures are needed (let alone replaced signatures).

IFF we can determine 'new revisions' and 'new signatures' without full graph access then we can scale acceptable for push and pull.

Ghosts are revisions which are not present in a particular repository. Filling ghosts refers to removing ghosts in the target repository when the ghost is present in the source repository. Filling ghosts can be either an explicit or implicit action. The common case is no ghosts.

#### 1.3.2.3.1.1 Set synchronisation approaches

A set synchronisation approach is one which synchronises two sets

without regard for innate structure. This can be very efficient but requires adding a new node to be processed with every commit. Caching of the results of the various set based syncs I've seen is possible but because the data structures look different depending on the tip revision being synced up to the cache needs to be very complex. I recommend not using such an approach for the common case pull because of the failure to scale. We can use such an approach for synchronisation of new signatures and ghosts, which should be an explicit option in both cases.

#### 1.3.2.3.1.2 DAG synchronisation approaches

A DAG based approach to synchronistion is one that uses the DAG structure to determine the difference in present nodes. It can as a result operate from the tip of the DAG backwards. A dag based approach should allow incremental access to data and not require a full-graph scan for incremental operations.

#### 1.3.2.3.1.3 File level scaling

We should read roughly as much of the revision level graph as is needed from each repository to determine the node difference. If requested we should perform a detailed scan to pick up ghost revisions and revisions which have had signatures added. This should not be the default as it requires full history access in both cases.

Expected file IO and access pattern:

- Common case: repo with many branches of one project, to the same.
  - 1. Source and Target branch tips read.
  - 2. Find the tip of each branch in their repo (will require reading some of the revision graph but is typically near

the end of the graph).

- 3. Read and parse increasing amounts of the revision graph until one is found to be a subset of the other, or a complete list of revisions to be transmitted is created.
- Uncommon cases:
  - 1. Repositories with many projects or branches which are very old may require reading a lot of unrelated graph data.
  - 1. Initial push/pull scenarios should not require reading an entire graph.

#### 1.3.2.3.1.4 API scaling

- 1. Get branch tips.
- 2. Determine one sided graph difference. To avoid obtaining a full graph over the wire this needs to be done without reference to the full graph, and with some logarthmic scaling algorithm. There are several already available for this.

With ghost and new-signature detection:

- File IO access pattern will read the entire graph on the 'target' side if no ghosts are present then stop, otherwise seek the new revisions on the source side with the regular algorithm and also explicitly search for the ghost points from the target; plus a set difference search is needed on signatures.
- Semantic level can probably be tuned, but as its also complex I suggest deferring analysis for optimal behaviour of this use case.

## 1.3.2.3.2 Data reading

When transferring information about a revision the graph of data for the revision is walked: revision -> inventory, revision -> matching signature, inventory -> file ids:revision pairs.

#### 1.3.2.3.2.1 File level scaling

As we're reading already committed data, as long as nothing is mutating data on disk reading should be race free. We will:

- read each revision object
- read the matching inventory delta
- attempt to read a signature object
- parse the inventory delta
- read the fileid:revisionid compressed chunk for each line in the inventory delta

Theres no point validating that the data read is valid, as transmission through to the client writing the data might invalidate it; we need to validate before we write.

#### 1.3.2.3.2.2 API scaling

Given that we have established the revisions needed, a single API call should suffice to obtain all data; the API should present the data in such an order that it can be validated as it arrives and thus not require large scale buffering on disk. Specifically each item of data should be validatable (e.g. for some file data we want the fileid:revisionid:validationhash + content).

## 1.3.2.3.3 Data Verification and writing

New data written to a repository should be completed intact when it is made visible. This suggests that either all the data for a revision must be made atomically visible (e.g. by renaming a single file) or the leaf nodes of the reference graph must become visible first. Data is referred to via the following graph: revision -> revision revision -> signature revision -> inventory inventory -> fileid:revisionid fileid:revisionid -> fileid:revisionid

Data is verifiable via a different ordering: signature -> revision -> inventory -> fileid:revisionid texts.

We dont gpg verify each revision today; this analysis only speaks to hash verification of contents.

To validate a revision we need to validate the data it refers to. But to validate the contents of a revision we need the new texts in the inventory for the revision - to check a fileid:revisionid we need to know the expected sha1 of the full text and thus also need to read the delta chain to construct the text as we accept it to determine if its valid. Providing separate validators for the chosen representation would address this. e.g. For an inventory entry FILEID:REVISIONID we store the validator of the full text :SHA1:. If we also stored the validator of the chosen disk representation (:DELTASHA1:) we could validate the transmitted representation without expanding the delta in the common case. If that failed we could expand the delta chain and try against the full text validator, and finally fail. As different delta generators might generate different deltas, :DELTASHA1: should not become part of the revision validator, only the inventory disk encoding. In a related manner a transmission format that allowed cheap validation of content without applying locally stored deltas would be advantageous because no local reads would be incurred to validate new content. For instance, always sending a full text for any file, possibly with a delta-chain when transmitting multiple revisionids of the file, would allow this. (git pack-files have this property).

#### 1.3.2.3.3.1 Overview summary

A single-file local format would allow safe atomic addition of data while allowing optimisal transmission order of data. Failing this the validation of data should be tuned to not require reading local texts during data addition even in the presence of delta chains. We should have transmission-validators separate from content validators that allow validation of the delta-transmitted form of objects.

#### 1.3.2.3.3.2 File level scaling

- Every new file text requires transmission and local serialisation.
- Every commit requires transmission and storage of a revision, signature and inventory.

Thus 4000 commits to a 50000 path tree of 10 files on averages requires (with knits) between 26 writes (2\*(3+10)) and 80006 (2\*(4000\*10 + 3)) writes. In all cases there are 4000 \* 13 distinct objects to record.

Grouping data by fileid, content and metadata, gives the figures above. Data grouping:

- File per full identifier (fileid:revisionid:meta|content): 104000
- Delta-chain per object: object id count \* constant overhead per object id (26 -> 80006)
- Collation/pack file: 1

Performance for these depends heavily on implementation:

- Using full ids we could name by validator or by id, giving best performance that depends on either receiving data in validator order or in id order.
- using delta-chain per object we get least seek overhead and syscall overhead if we recieve in topological order within the object id, and object ids in lexical order.
- Using a collation/pack file we can stream it into place and validate as we go, giving near ideal performance.

1.3.2.3.3.3 API scaling

The api for writing new data recieved over the network will need to be geared to the transmission and local storage method. What we need is for the transmission method to reasonably closely match the desired write ordering locally. This suggests that once we decide on the best local storage means we should design the api.

take N commits from A to B, if B is local then merge changes into the tree. copy ebough data to recreate snapshots avoid ending up wth corrupt/bad data

#### 1.3.2.4 Notes from London

1. setup

look at graph of revisions for ~N comits to deretmine eligibility for if preserve mainline is on, check LH only

identify objects to send that are not on the client repo

- revision may be proportional to the graph
- inventory proportional to work
- texts proportional to work
- signatures ???
- 1. data transmission
  - send data proportional to the new information
  - validate the data:
    - 1. validate the sha1 of the full text of each transmitted text.
    - 2. validate the sha1:name mapping in each newly referenced inventory item.
    - 3. validate the sha1 of the XML of each inventory against the revision. this is proportional to tree size and must be fixed

- 1. write the data to the local repo. The API should output the file texts needed by the merge as by product of the transmission
- 2. tree application

Combine the output from the transmission step with additional 'new work data' for anything already in the local repository that is new in this tree. should write new files and stat existing files proportional to the count of the new work and the size of the full texts.

# 1.3.3 Add

Add is used to recursively version some paths supplied by the user. Paths that match ignore rules are not versioned, and paths that become versioned are versioned in the nearest containing bzr tree. Currently we only do this within a single tree, but perhaps with nested trees this should change.

#### 1.3.3.1 Least work we can hope to perform

- Read a subset of the full versioned paths data for the tree matching the scope of the paths the user supplied.
- Seek once to each directory within the scope and readdir its contents.
- Probe if each directory is a child tree to avoid adding data for paths within a child tree.
- Calculate the ignored status for paths not previously known to be ignored
- Write data proportional to the newly versioned file count to record their versioning.
- Assign a fileid for each path (so that merge –uncommitted can work immediately)

Optionally:

- Print the ignore rule for each ignored path in the scope.
- Print the path of each added file.
- Print the total count of ignored files within the scopes.
- Record the result of calculating ignored status for ignored files. (proportional to the number we actually calculate).

## 1.3.3.2 Per file algorithm

- 1. If the path is versioned, and it is a directory, push onto the recurse stack.
- 2. If the path is supplied by the user or is not ignored, version it, and if a directory, push onto the recurse stack. Versioning the path may require versioning the paths parents.
- 3. Output or otherwise record the ignored rule as per the user interface selected.

# 1.3.4 Commit Performance Notes

- 1.3.4.1 Changes to commit
- 1.3.4.2 Commit: The Minimum Work Required
- 1.3.4.3 Commit vs Status
- 1.3.4.4 Avoiding Work: Smarter Change Detection
- 1.3.4.5 Avoiding Work: Better Layering
- 1.3.4.6 Avoiding work: avoiding reading parent data
- 1.3.4.7 Code structure
- 1.3.4.8 Complications of commit
- 1.3.4.9 Interface stack
- 1.3.4.10 Branch->Tree interface
- 1.3.4.11 Information from the tree to repository
- 1.3.4.12 Information from the repository to the tree
- 1.3.4.13 Selective commit
- 1.3.4.14 Common commit code
- 1.3.4.15 Order of traversal
- 1.3.4.16 Open question: per-file graphs

## 1.3.4.1 Changes to commit

We want to improve the commit code in two phases.

Phase one is to have a better separation from the format-specific logic, the user interface, and the general process of committing.

Phase two is to have better interfaces by which a good workingtree format can efficiently pass data to a good storage format. If we get phase one right, it will be relatively easy and non-disruptive to bring this in.

#### 1.3.4.2 Commit: The Minimum Work Required

Here is a description of the minimum work that commit must do. We want to make sure that our design doesn't cost too much more than this minimum. I am trying to do this without making too many assumptions about the underlying storage, but am assuming that the ui and basic architecture (wt, branch, repo) stays about the same.

The basic purpose of commit is to:

- 1. create and store a new revision based on the contents of the working tree
- 2. make this the new basis revision for the working tree

We can do a selected commit of only some files or subtrees.

The best performance we could hope for is: - stat each versioned selected working file once - read from the workingtree and write into the repository any new file texts - in general, do work proportional to the size of the shape (eg inventory) of the old and new selected trees, and to the total size of the modified files

In more detail:

1.0 - Store new file texts: if a versioned file contains a new text there is no avoiding storing it. To determine which ones have changed we must go over the workingtree and at least stat each file. If the file is modified since it was last hashed, it must be read in. Ideally we would read it only once, and either notice that it has not changed, or store it at that point.

On the other hand we want new code to be able to handle files that are larger than will fit in memory. We may then need to read each file up to two times: once to determine if there is a new text and calculate its hash, and again to store it.

1.1 - Store a tree-shape description (ie inventory or similar.) This describes the non-file objects, and provides a reference from the Revision to the texts within it.

1.2 - Generate and store a new revision object.

1.3 - Do delta-compression on the stored objects. (git notably does not do this at commit time, deferring this entirely until later.) This requires finding the appropriate basis for each modified file: in the current scheme we get the file id, last-revision from the dirstate, look into the knit for that text, extract that text in total, generate a delta, then store that into the knit. Most delta operations are  $O(n^{**}2)$  to  $O(n^{**}3)$  in the size of the modified files.

1.4 - Cache annotation information for the changes: at the moment this is done as part of the delta storage. There are some flaws in that approach, such as that it is not updated when ghosts are filled, and the annotation can't be re-run with new diff parameters.

2.1 - Make the new revision the basis for the tree, and clear the list of parents. Strictly this is all that's logically necessary, unless the working tree format requires more work. The dirstate format does require more work, because it caches the parent tree data for each file within the working tree data. In practice this means that every commit rewrites the entire dirstate file - we could try to avoid rewriting the whole file but this may be difficult because variable-length data (the last-changed revision id) is inserted into many rows.

The current dirstate design then seems to mean that any commit of a single file imposes a cost proportional to the size of the current workingtree. Maybe there are other benefits that outweigh this. Alternatively if it was fast enough for operations to always look at the original storage of the parent trees we could do without the cache.

2.2 - Record the observed file hashes into the workingtree control files. For the files that we just committed, we have the information to store a valid hash cache entry: we know their stat information and the sha1 of the file contents. This is not strictly necessary to the speed of commit, but it will be useful later in avoiding reading those files, and the only cost of doing it now is writing it out.

In fact there are some user interface niceties that complicate this:

3 - Before starting the commit proper, we prompt for a commit message and in that commit message editor we show a list of the files that will be committed: basically the output of bzr status. This is basically the same as the list of changes we detect while storing the commit, but because the user will sometimes change the tree after opening the commit editor and expect the final state to be committed I think we do have to look for changes twice. Since it takes the user a while to enter a message this is not a big problem as long as both the status summary and the commit are individually fast.

4 - As the commit proceeds (or after?) we show another status-like summary. Just printing the names of modified files as they're stored would be easy. Recording deleted and renamed files or directories is more work: this can only be done by reference to the primary parent tree and requires it be read in. Worse, reporting renames requires searching by id across the entire parent tree. Possibly full reporting should be a default-off verbose option because it does require more work beyond the commit itself.

5 - Bazaar currently allows for missing files to be automatically marked as removed at the time of commit. Leaving aside the ui consequences, this means that we have to update the working inventory to mark these files as removed. Since as discussed above we always have to rewrite the dirstate on commit this is not substantial, though we should make sure we do this in one pass, not two. I have previously proposed to make this behaviour a non-default option.

We may need to run hooks or generate signatures during commit, but they don't seem to have substantial performance consequences.

If one wanted to optimize solely for the speed of commit I think hashaddressed file-per-text storage like in git (or bzr 0.1) is very good. Remarkably, it does not need to read the inventory for the previous revision. For each versioned file, we just need to get its hash, either by reading the file or validating its stat data. If that hash is not already in the repository, the file is just copied in and compressed. As directories are traversed, they're turned into texts and stored as well, and then finally the revision is too. This does depend on later doing some delta compression of these texts.

Variations on this are possible. Rather than writing a single file into the repository for each text, we could fold them into a single collation or pack file. That would create a smaller number of files in the repository, but looking up a single text would require looking into their indexes rather than just asking the filesystem.

Rather than using hashes we can use file-id/rev-id pairs as at

present, which has several consequences pro and con.

## 1.3.4.3 Commit vs Status

At first glance, commit simply stores the changes status reports. In fact, this isn't technically correct: commit considers some files modified that status does not. The notes below were put together by John Arbash Meinel and Aaron Bentley in May 2007 to explain the finer details of commit to Ian Clatworthy. They are recorded here as they are likely to be useful to others new to Bazaar ...

- 1. **Unknown files have a different effect.** With –no-strict (the default) they have no effect and can be completely ignored. With –strict they should cause the commit to abort (so you don't forget to add the two new test files that you just created).
- 2. **Multiple parents.** 'status' always compares 2 trees, typically the last-committed tree and the current working tree. 'commit' will compare more trees if there has been a merge.
  - a. The "last modified" property for files. A file may be marked as changed since the last commit, but that change may have come in from the merge, and the change could have happened several commits back. There are several edge cases to be handled here, like if both branches modified the same file, or if just one branch modified it.
  - b. The trickier case is when a file appears unmodified since last commit, but it was modified versus one of the merged branches. I believe there are a few ways this can happen, like if a merged branch changes a file and then reverts it back (you still update the 'last modified' field). In general, if both sides disagree on the 'last-modified' flag, then you need to generate a new entry pointing 'last-modified' at this revision (because you are resolving the differences between the 2 parents).

- 3. Automatic deletion of 'missing' files. This is a point that we go back and forth on. I think the basic idea is that 'bzr commit' by default should abort if it finds a 'missing' file (in case that file was renamed rather than deleted), but 'bzr commit –auto' can add unknown files and remove missing files automatically.
- 4. **sha1 for newly added files.** status doesn't really need this: it should only care that the file is not present in base, but is present now. In some ways commit doesn't care either, since it needs to read and sha the file itself anyway.
- 5. **Nested trees.** status doesn't recurse into nested trees, but commit does. This is just because not all of the nested-trees work has been merged yet.

A tree-reference is considered modified if the subtree has been committed since the last containing-tree commit. But commit needs to recurse into every subtree, to ensure that a commit is done if the subtree has changed since its last commit. \_iter\_changes only reports on tree-references that are modified, so it can't be used for doing subtree commits.

## 1.3.4.4 Avoiding Work: Smarter Change Detection

Commit currently walks through every file building an inventory. Here is Aaron's brain dump on a better way ...

\_iter\_changes won't tell us about tree references that haven't changed, even if those subtrees have changed. (Unless we ask for unchanged files, which we don't want to do, of course.)

There is an iter\_references method, but using it looks just as expensive as calling kind().

I did some work on updating commit to use iter\_changes, but found for multi-parent trees, I had to fall back to the slow inventory comparison approach.

Really, I think we need a call akin to iter\_changes that handles multiple parents, and knows to emit entries when InventoryEntry.revision is all that's changed.

#### 1.3.4.5 Avoiding Work: Better Layering

For each file, commit is currently doing more work than it should. Here is John's take on a better way ...

Note that "\_iter\_changes" *does* have to touch every path on disk, but it just can do it in a more efficient manner. (It doesn't have to create an InventoryEntry for all the ones that haven't changed).

I agree with Aaron that we need something a little different than \_iter\_changes. Both because of handling multiple parents, as well as we don't want it to actually read the files if we have a stat-cache miss.

Specifically, the commit code *has* to read the files because it is going to add the text to the repository, and we want it to compute the sha1 at *that* time, so we are guaranteed to have the valid sha (rather than just whatever the last cached one was). So we want the code to return 'None' if it doesn't have an up-to-date sha1, rather than reading the file and computing it, just before it returns it to the parent.

The commit code (0.16) should really be restructured. It's layering is pretty wrong.

Specifically, calling "kind()" requires a stat of the file. But we have to do a stat to get the size/whether the record is up-to-date, etc. So we really need to have a "create\_an\_up\_to\_date\_inventory()" function. But because we are accessing every object on disk, we want to be working in tuples rather than Inventory objects. And because DirState already has the parent records next to the current working

inventory, it can do all the work to do really fast comparison and throw-away of unimportant records.

The way I made "bzr status" fast is by moving the 'ignore this record' ability as deep into the stack as I could get. Status has the property that you don't care about most of the records, just like commit. So the sooner you can stop evaluating the 99% that you don't care about, the less work you do.

#### 1.3.4.6 Avoiding work: avoiding reading parent data

We would like to avoid the work of reading any data about the parent revisions. We should at least try to avoid reading anything from the repository; we can also consider whether it is possible or useful to hold less parent information in the working tree.

When a commit of selected files is requested, the committed snapshot is a composite of some directories from the parent revision and some from the working tree. In this case it is logically necessary to have the parent inventory information.

If file last-change information or per-file graph information is stored then it must be available from the parent trees.

If the Branch's storage method does delta compression at commit time it may need to retrieve file or inventory texts from the repository.

It is desirable to avoid roundtrips to the Repository during commit, particularly because it may be remote. If the WorkingTree can determine by itself that a text was in the parent and therefore should be in the Repository that avoids one roundtrip per file.

There is a possibility here that the parent revision is not stored, or not correctly stored, in the repository the tree is being committed into, and so the committed tree would not be reconstructable. We could check that the parent revision is present in the inventory and rely on the invariant that if a revision is present, everything to reconstruct it will be present too.

#### 1.3.4.7 Code structure

Caller starts a commit

```
>>> Branch.commit(from_tree, options)
```

This creates a CommitBuilder object matched to the Branch, Repository and Tree. It can vary depending on model differences or by knowledge of what is efficient with the Repository and Tree. Model differences might include whether no-text-change merges need to be reported, and whether the

The basic CommitBuilder.commit structure can be

- 1. Ask the branch if it is ready to commit (up to date with master if any.)
- 2. Ask the tree if it is ready to commit to the branch (up to date with branch?), no conflicts, etc
- 3. Commit changed files; prototype implementation:
  - a. Ask the working tree for all committable files; for each it should return the per-file parents, stat information, kind, etc.
  - b. Ask the repository to store the new file text; the repository should return the stored sha1 and new revision id.
- 4. Commit changed inventory
- 5. Commit revision object

## 1.3.4.8 Complications of commit

Bazaar (as of 0.17) does not support selective-file commit of a merge; this could be done if we decide how it should be recorded - is this to be stored as an overall merge revision; as a preliminary non-merge revisions; or will the per-file graph diverge from the revision

graph.

There are several checks that may cause the commit to be refused, which may be activated or deactivated by options.

- presence of conflicts in the tree
- presence of unknown files
- the working tree basis is up to date with the branch tip
- the local branch is up to date with the master branch, if there is one and –local is not specified
- an empty commit message is given,
- a hook flags an error
- a "pointless" commit, with no inventory changes

Most of these require walking the tree and can be easily done while recording the tree shape. This does require that it be possible to abort the commit after the tree changes have been recorded. It could be ok to either leave the unreachable partly-committed records in the repository, or to roll back.

Other complications:

- when automatically adding new files or deleting missing files during commit, they must be noted during commit and written into the working tree at some point
- refuse "pointless" commits with no file changes should be easy by just refusing to do the final step of storing a new overall inventory and revision object
- heuristic detection of renames between add and delete (out of scope for this change)
- pushing changes to a master branch if any
- running hooks, pre and post commit
- prompting for a commit message if necessary, including a list of the changes that have already been observed
- if there are tree references and recursing into them is enabled,

then do so

Commit needs to protect against duplicated file ids

Updates that need to be made in the working tree, either on conclusion of commit or during the scan, include

- Changes made to the tree shape, including automatic adds, renames or deletes
- For trees (eg dirstate) that cache parent inventories, the old parent information must be removed and the new one inserted
- The tree hashcache information should be updated to reflect the stat value at which the file was the same as the committed version, and the content hash it was observed to have. This needs to be done carefully to prevent inconsistencies if the file is modified during or shortly after the commit. Perhaps it would work to read the mtime of the file before we read its text to commit.

#### 1.3.4.9 Interface stack

The commit api is invoked by the command interface, and copies information from the tree into the branch and its repository, possibly updating the WorkingTree afterwards.

The command interface passes:

- a commit message (from an option, if any),
- or an indication that it should be read interactively from the ui object;
- a list of files to commit
- an option for a dry-run commit
- verbose option, or callback to indicate
- timestamp, timezone, committer, chosen revision id
- config (for what?)

- option for local-only commit on a bound branch
- option for strict commits (fail if there are unknown or missing files)
- option to allow "pointless" commits (with no tree changes)

(This is rather a lot of options to pass individually and just for code tidyness maybe some of them should be combine into objects.)

>>> Branch.commit(from\_tree, message, files\_to\_commit, ...)

There will be different implementations of this for different Branch classes, whether for foreign branches or Bazaar repositories using different storage methods.

Most of the commit should occur during a single lockstep iteration across the workingtree and parent trees. The WorkingTree interface needs to provide methods that give commit all it needs. Some of these methods (such as answering the file's last change revision) may be deprecated in newer working trees and there we have a choice of either calculating the value from the data that is present, or refusing to support commit to newer repositories.

For a dirstate tree the iteration of changes from the parent can easily be done within its own iter\_changes.

Dirstate inventories may be most easily updated in a single operation at the end; however it may be best to accumulate data as we proceed through the tree rather than revisiting it at the end.

Showing a progress bar for commit may not be necessary if we report files as they are committed. Alternatively we could transiently show a progress bar for each directory that's scanned, even if no changes are observed.

This needs to collect a list of added/changed/removed files, each of which must have its text stored (if any) and containing directory

updated. This can be done by calling Tree.\_iter\_changes on the source tree, asking for changes

In the 0.17 model the commit operation needs to know the per-file parents and per-file last-changed revision.

(In this and other operations we must avoid having multiple layers walk over the tree separately. For example, it is no good to have the Command layer walk the tree to generate a list of all file ids to commit, because the tree will also be walked later. The layers that do need to operate per-file should probably be bound together in a perdirblock iterator, rather than each iterating independently.)

#### 1.3.4.10 Branch->Tree interface

The Branch commit code needs to ask the Tree what should be committed, in terms of changes from the parent revisions. If the Tree holds all the necessary parent tree information itself it can do it single handed; otherwise it may need to ask the Repository for parent information.

This should be a streaming interface, probably like iter\_changes returning information per directory block.

The interface should not return a block for directories that are recursively unchanged.

The tree's idea of what is possibly changed may be more conservative than that of the branch. For example the tree may report on merges of files where the text is identical to the parents: this must be recorded for Bazaar branches that record per-file ancestry but is not necessary for all branches. If the tree is responsible for determining when directories have been recursively modified then it will report on all the parents of such files. There are several implementation options: 1. Return all files and directories the branch might want to commit, even if the branch ends up taking no action on them.

2. When starting the iteration, the branch can specify what type of change is considered interesting.

Since these types of changes are probably (??) rare compared to files that are either completely unmodified or substantially modified, the first may be the best and simplest option.

The branch needs to build an inventory to commit, which must include unchanged files within changed directories. This should be returned from the working tree too. Repositories that store perdirectory inventories will want to build and store these from the lowest directories up. For 0.17 format repositories with an all-in-one inventory it may be easiest to accumulate inventory entries in arbitrary order into an in-memory Inventory and then serialize it.

It ought to be possible to commit any Tree into a Branch, without requiring a WorkingTree; the commit code should cope if the tree is not interested in updating hashcache information or does not have a last\_revision.

## 1.3.4.11 Information from the tree to repository

The main things the tree needs to tell the Branch about are:

• A file is modified from its parent revision (in text, permissions, other), and so its text may need to be stored.

Files should also be reported if they have more than one unique parent revision, for repositories that store per-file graphs or last-change revisions. Perhaps this behaviour should be optional.

**XXX:** are renames/deletions reported here too?

• The complete contents of a modified directory, so that its inventory text may be stored. This should be done after all the contained files and directories have been reported. If there are unmodified files, or unselected files carried through from

XXX: Actually perhaps not grouped by directory, but rather grouped appropriately for the shape of inventory storage in the repository.

In a zoomed-in checkout the workingtree may not have all the shape data for the entire tree.

- A file is missing could cause either automatic removal or an aborted commit.
- Any unknown files can cause automatic addition, abortion of a strict commit, or just reporting.

#### 1.3.4.12 Information from the repository to the tree

After the commit the tree needs to be updated to the new revision. Some information which was accumulated during the commit must be made available to the workingtree. It's probably reasonable to hold it all in memory and allow the workingtree to get it in whatever order it wants.

- A list of modified entries, and for each one:
  - The stat values observed when the file was first read.
  - The hash of the committed file text.
  - The file's last-change revision, if appropriate.

This should include any entries automatically added or removed.

This might be construed as an enhanced version of set\_parent\_trees. We can avoid a stat on each file by using the

value that was observed when it was first read.

## 1.3.4.13 Selective commit

For a partial commit the directory contents may need to contain a mix of entries from the working tree and parent trees. This code probably shouldn't live in a specific tree implementation; maybe there should be a general filter that selects paths from one tree into another?

However, the tree walking code does probably need to know about selected paths to avoid examining unselected files or directories.

We never refuse selective file commits (except of merges).

#### 1.3.4.14 Common commit code

What is common to all commit implementations, regardless of workingtree or repository format?

- Prompting for a commit message?
- Strictness/conflict checks?
- Auto add/remove?

How should this be separated?

## 1.3.4.15 Order of traversal

For current and contemplated Bazaar storage formats, we can only finally commit a directory after its contained files and directories have been committed.

The dirstate workingtree format naturally iterates by directory in order by path, yielding directories before their contents. This may also be the most efficient order in which to stat and read the files. One option would be to construe the interface as a visitor which reports when files are detected to be changed, and also when directories are finished.

#### 1.3.4.16 Open question: per-file graphs

**XXX:** If we want to retain explicitly stored per-file graphs, it would seem that we do need to record per-file parents. We have not yet finally settled that we do want to remove them or treat them as a cache. This api stack is still ok whether we do or not, but the internals of it may change.

## 1.3.5 diff Performance Analysis

- 1.3.5.1 Minimal Work
  - 1.3.5.1.1 Reuse of historical comparisons
  - 1.3.5.1.2 Historical Tree Against Historical Tree
  - 1.3.5.1.3 Basis Against Historical Tree
  - 1.3.5.1.4 Basis Against Basis
  - 1.3.5.1.5 Working Tree Against Basis
  - 1.3.5.1.6 Working Tree Against Historical Tree
  - 1.3.5.1.7 Working Tree Against Working Tree
- 1.3.5.2 API Changes
- 1.3.5.3 Storage considerations

#### 1.3.5.1 Minimal Work

#### 1.3.5.1.1 Reuse of historical comparisons

A significant part of the work done by diff is sequence matching. This scales  $O(n^2)$  with the number of lines in the file. Therefore, it is worthwile to avoid content comparisons as much as possible.

Our current knit format contains content comparisons, and this data can be converted into lists of matching blocks. Other future formats such as mpdiff may also support such conversion. So it is possible to reuse past comparisons.

It is also possible to combine sequential comparisons. So given a comparison of "foo" to "bar", and "bar" to "baz", it is possible to derive a comparison of "foo" to "baz".

Reuse of historical comparisons will scale with the number of uncommon build-parents between the two historical revisions. This will typically be proportional to the amount of change that the file has undergone. Therefore, in the common case, reuse of historical comparisons will scale with the amount of change.

The downside of such reuse is that it ties the comparison to the historical data. But given the performance improvement, it seems to be worth consideration. Fresh comparisons can be performed if the user requests them.

It may also be possible to accelerate comparisons by including annotation data, thus increasing the number of unique lines.

## 1.3.5.1.2 Historical Tree Against Historical Tree

This operation should be strictly proportional to the amount of change, because a comparison has already been done at commit time. Achieving that performance requires the committed data to be properly structured, so that the comparison can be extracted and combined with other comparisons. This comparision extraction should be possible at the inventory and file-content levels.

Minimum work:

- 1. Extract and combine inventory comparisons
- 2. Extract and combine text comparisions for modified texts

## 1.3.5.1.3 Basis Against Historical Tree

This is another case of Historical Tree Against Historical Tree.

1.3.5.1.4 Basis Against Basis

This is another case of Historical Tree Against Historical Tree.

## 1.3.5.1.5 Working Tree Against Basis

This must scale with the number of versioned files, unless the user indicates that only certain files should be compared.

Performance can be further improved by caching comparisons to avoid repeating them. Caching could potentially be performed by diff and perhaps by merge. Merge is aware of the relationship of a text merge's result to the THIS value, and the THIS value is generally the basis value. So the comparison is latent, but present. The only issue is extracting it.

The cache could be indexed by sha1sum pairs. It could also be indexed by file-id, to facilitate removal of stale data.

Minimum work:

- 1. Scan working tree for modified files
- 2. Retrieve cached comparisons
- 3. Perform comparisons on files with no cached comparisons
- 4. Cache comparisons for files with no cached comparisons

#### 1.3.5.1.6 Working Tree Against Historical Tree

This can be structured as a comparison of working tree against basis tree, followed by basis tree against historical tree. Therefore, it combines the performance characteristics of "Working Tree Against Basis" with "Basis Against Historical Tree".

#### 1.3.5.1.7 Working Tree Against Working Tree

This can be structured as two comparisons against basis, and one comparison of basis against basis. Its performance is therefore similar to Working Tree Against Historical Tree.

## 1.3.5.2 API Changes

Desired API:

• Tree.get\_comparision(file\_id, tree)

This probably entails:

- WorkingTree.store\_comparison(file\_id, revision\_id, sha1, comparison)
- WorkingTree.get\_comparison(file\_id, revision\_id, sha1)
- Repository.get\_comparision(file\_id, revision\_id, revision\_id)
- merge\_comparisions(comparison, comparision)

## 1.3.5.3 Storage considerations

It must be cheap (e.g. scale with number of intermediate revisions) to perform comparison of two historical texts. It must be cheap to perform comparison of the inventories of two historical trees.

# 1.3.6 Garbage Collection

Garbage collection is used to remove data from a repository that is no longer referenced.

Generally this involves locking the repository and scanning all its branches then generating a new repository with less data.

## 1.3.6.1 Least work we can hope to perform

- Read all branches to get initial references tips + tags.
- Read through the revision graph to find unreferenced revisions. A cheap HEADS list might help here by allowing comparison of the initial references to the HEADS - any unreferenced head is garbage.
- Walk out via inventory deltas to get the full set of texts and signatures to preserve.
- Copy to a new repository
- Bait and switch back to the original
- Remove the old repository.

A possibility to reduce this would be to have a set of grouped 'known garbage free' data - 'ancient history' which can be preserved in total should its HEADS be fully referenced - and where the HEADS list is deliberate cheap (e.g. at the top of some index).

possibly - null data in place without saving size.

## 1.3.7 Revert

Change users selected paths to be the same as those in a given revision making backups of any paths that bzr did not set the last contents itself.

#### 1.3.7.1 Least work we can hope to perform

We should be able to do work proportional to the scope the user is reverting and the amount of changes between the working tree and the revision being reverted to.

This depends on being able to compare unchanged subtrees without recursing so that the mapping of paths to revert to ids to revert can be done efficiently. Specifically we should be able to avoid getting the transitive closure of directory contents when mapping back to paths from ids at the start of revert.

One way this might work is to: for the selected scopes, for each element in the wt:

1. get hash tree data for that scope. 1. get 'new enough' hash data for the siblings of the scope: it can be out of date as long as its not older than the last move or rename out of that siblings scope. 1. Use the hash tree data to tune the work done in finding matching paths/ids which are different in the two trees.

For each thing that needs to change - group by target directory?

1. Extract new content. 1. Backup old content or replace-inplace (except windows where we move and replace).

# 1.3.8 The status command

The status command is used to provide a pithy listing of the changes between two trees. Its common case is between the working tree and the basis tree, but it can be used between any two arbitrary trees.

- 1.3.8.1 UI Overview
- 1.3.8.2 Ideal work for working tree to historical status
- 1.3.8.3 Locality of reference
- 1.3.8.4 Scaling observations

## 1.3.8.1 UI Overview

Status shows several things in parallel (for the paths the user supplied mapped across the from and to tree, and any pending merges in the to tree).

- 1. Single line summary of all new revisions the pending merges and their parents recursively.
- 2. Changes to the tree shape adds/deletes/renames.
- 3. Changes to versioned content kind changes and content changes.
- 4. Unknown files in the to tree.
- 5. Files with conflicts in the to tree.

#### 1.3.8.2 Ideal work for working tree to historical status

We need to do the following things at a minimum:

- 1. Determine new revisions the pending merges and history.
- 1. Retrieve the first line of the commit message for the new revisions.
- 1. Determine the tree differences between the two trees using the users paths to limit the scope, and resolving paths in the trees for any pending merges. We arguably don't care about tracking metadata for this only the value of the tree the user commited.
- 1. The entire contents of directories which are versioned when showing unknowns.
- 1. Whether a given unversioned path is unknown or ignored.
- 1. The list conflicted paths in the tree (which match the users path selection?)

Expanding on the tree difference case we will need to:

- 1. Stat every path in working trees which is included by the users path selection to ascertain kind and execute bit.
- 1. For paths which have the same kind in both trees and have

content, read that content or otherwise determine whether the content has changed. Using our hash cache from the dirstate allows us to avoid reading the file in the common case. There are alternative ways to achieve this - we could record a pointer to a revision which contained this fileid with the current content rather than storing the content's hash; but this seems to be a pointless double-indirection unless we save enough storage in the working tree. A variation of this is to not record an explicit pointer but instead define an implicit pointer as being to the lefthand-parent tree.

#### 1.3.8.3 Locality of reference

- We should stat files in the same directory without reading or statting files in other directories. That is we should do all the statting we intend to do within a given directory without doing any other IO, to minimise pressure on the drive heads to seek.
- We should read files in the same directory without reading or writing files in other directories and note this is separate to statting (file data is usually physically disjoint to metadata).

## 1.3.8.4 Scaling observations

- The stat operation clearly involves every versioned path in the common case.
- Expanding out the users path selection in a naive manner involves reading the entire tree shape information for both trees and for all pending-merge trees. (Dirstate makes this tolerably cheap for now, but we're still scaling extra-linearly.)
- The amount of effort required to generate tree differences between the working tree and the basis tree is interesting: with a tree-like structure and some generatable name for child nodes we use the working tree data to eliminate accessing or considering subtrees regardless of historival age. However, if we
have had to access the historical tree shape to perform path selection this rather reduces the win we can obtain here. If we can cause path expansion to not require historical shape access (perhaps by performing the expansion after calculating the tree difference for the top level of the selected path) then we can gain a larger win. This strongly suggests that path expansion and tree difference generation should be linked in terms of API.

# 1.3.9 Annotate

Broadly tries to ascribe parts of the tree state to individual commits.

There appear to be three basic ways of generating annotations:

If the annotation works by asking the storage layer for successive full texts then the scaling of this will be proportional to the time to diff throughout the history of thing being annotated.

If the annotation works by asking the storage layer for successive deltas within the history of the thing being annotated we believe we can make it scale broadly proportional to the depth of the tree of revisions of the annotated object.

If the annotation works by combining cached annotations such that creating a full text recreates annotations for it then it will scale with the cost of obtaining that text.

Generally we want our current annotations but it would be nice to be able to do whitespace annotations and potentially other diff based annotations.

Some things to think about:

• Perhaps multiparent deltas would allow us to not store the cached annotations in each delta without losing performance or accuracy.

# 1.3.10 Scaling analysys of Merge

- 1. Fetch revisions O(a)
- 2. Common Ancestor [O(b)] O(h)
- 3. Calculate tree merge O(c) [+ O(b) + O(d)] + O(i)
  - text merge O(e \* e \* f) + O(b)
- 4. Find filesystem conflicts O(c)
- 5. Resolve filesystem conflicts O(g)
- 6. Apply changes O(c) + O(log(d))
- 7. Set pending merges O(1)
- 8. Print conflicts O(g)
- 9. Print changes O(c)
- a: revisions missing from repo:
- **b:** nodes in the revision graph:
- c: files that differ between base and other:
- d: number of files in the tree
- e: number of lines in the text
- f: number of files requiring text merge
- **g:** number of conflicts (g <= c)
- h: humber of uncommon ancestors
- i: number of revisions between base and other

#### 1.3.10.1 Needs

- Access to revision graph proportional to number of revisions read
- Access to changed file metadata proportional to number of changes and number of intervening revisions.
- O(1) access to fulltexts

#### 1.3.10.2 Notes

Multiparent deltas may offer some nice properties for performance of annotation based merging.

# 1.3.11 Bundle Creation

- 1. Find common ancestor [O(a)] O(b)
- 2. Emit bundle [O(a)] O(b) O(h)

Per revision

- 1. emit metadata O(1)
- 2. emit changes for files
  - 1. find changed files [O(c)] O(f)
  - 2. emit file metadata O(d)
  - 3. emit diff [O(e \* e) \* O(f) + O(h)] O(i)
  - 4. base64 encode O(g)

#### 3. emit overal diff (or maybe do interdiff) O(e \* e) \* O(f)

- a: nodes in revision graph
- **b:** number of descendants of common ancestor
- c: number of files in the tree
- d: length of metadata
- e: number of lines
- f: number of modified files
- g: length of diff
- h: nodes in knit graph of modified files
- i: length of stored diff

### 1.3.11.1 Needs

- Improved common ancestor algorithm
- Access to partial revision graph proportional to relevant revisions
- Access to changed files proportional to number of change files

and intervening revisions

- Use knit deltas without recomputing
- Access to knit deltas in O(1) time
- Access to snapshots in O(1) amortized time
- All snapshots must have knit deltas

### 1.3.12 Uncommit Performance Notes

### 1.3.12.1 Specification of uncommit

uncommit removes revisions from the head of a branch. (By default, only the very latest revision is removed, but optionally more can be taken.) Uncommit does not affect the repository (garbage collection is a separate step and not done by default). The working tree is not logically modified (revert is a different operation), except as described below about merges.

Uncommit can be performed on either a branch or a working tree (and implicitly its branch.)

If the uncommitted revisions includes one or more merges, after the uncommit those revisions are in the working tree's list of pending merges, because their tree changes are still present in the tree.

For a bound branch, uncommit fails unless the local branch is up to date.

## 1.3.13 Missing

Missing is used to find out the differences between the current branch and another branch.

The performance analysis itself brings no further points than the incremental-push-pull one.

More importantly, the UI have been considered not optimal: missing finds and displays the differences between two branches, presenting the revisions that are not common to both branches as two sets:

- the revisions that are present only in the current branch,
- the revisions that are present only in the other branch.

A quick and dirty survey indicates that most of the users are interested in only one set of revisions at a time.

From a performance point of view, it may be more appropriate to calculate only the set the user is asking for.

It has been proposed that the missing command be deprecated in favor of a –dry-run option for the push, pull, merge commands.

In the mean time, the missing command stays interesting as it provides an easy way to test, measure and optimize graph differences processing.

# 1.4 Subsystem designs

# 1.4.1 Directory fingerprints

- 1.4.1.1 Introduction
- 1.4.1.2 Use-case oriented APIs
  - $\circ$  1.4.1.2.1 commit
  - 1.4.1.2.2 log
- 1.4.1.3 Open questions
- 1.4.1.4 Conclusions
- 1.4.1.5 Design changes
- 1.4.1.6 API changes

#### 1.4.1.1 Introduction

The basic idea is that for a directory in a tree (committed or otherwise), we will have a single scalar value. If these values are the same, the contents of the subtree under that directory are necessarily the same.

This is intended to help with these use cases, by allowing them to quickly skip over directories with no relevant changes, and to detect when a directory has changed:

- diff/status (both local trees and historical trees)
- merge
- log -v
- log on a directory
- commit

#### 1.4.1.2 Use-case oriented APIs

Most of this will be hidden behind the Tree interface. This should cover  $\log -v$ , diff, status, merge (and implicit merge during push, pull, update):

```
tree.iter_changes(other_tree)
tree.get_file_lines(file_id) # and get_file, get_file_text
```

#### 1.4.1.2.1 commit

Commit is similar to <u>iter\_changes</u>, but different because it needs to compare to all the trees. Commit currently needs to compare the working tree to all the parent trees, which is needed to update the last\_modified field and would be unnecessary if we removed that field (for both files and directories) and did not store per-file graphs. This would potentially speed up commit after merge.

Verbose commit also displays the merged files, which does require looking at all parents of files that aren't identical to the left-hand parent.

#### 1.4.1.2.2 log

Log is interested in two operations: finding the revisions that touched anything inside a directory, and getting the differences between consecutive revisions (possibly filtered to a directory):

```
find_touching_revisions(branch, file_id) # should be on Branch?
```

Log shows the revisions that merged a change. At the moment that is not included in the per-file graph, and it would also not be visible if the directories were hashed.

#### 1.4.1.3 Open questions

• Is this a good idea at all?

If changing a file changes all its parent directories up to the root it will cause more churn on commit. (We currently update the allin-one inventory, but only have to update one line of it.)

Every time a child changes, we'll get a new node in the perdirectory graph. This is generally useful: it allows bzr log to do the default mode easily, which is to show all changes under that directory. The less common operation, log --no-recursive is still possible by looking only at when the directory itself was renamed, added or removed. (That is what the directory graph describes in bzr 0.18 and it is rarely useful.)

• Should these be hashes or revision ids or something else?

Pros of using hashes: hashes are easy to generate by a foreign branch plugin (e.g. bzr-svn). They don't need to get recursive last-changed from the foreign branch, or to walk back through history. They just need the relevant directory state, which any system we support can answer.

Hashes converge: if you modify and then modify back, you get the same hash. This is a pro because you can detect that there were ultimately no significant changes. And also a con: you cannot use these hashes to form a graph because they get cycles.

• Are the values unique across the whole tree, or only when comparing different versions of the same object?

If we use last-changed revisions, then they will be very not unique across the whole tree. To look up the contents, you must pass a composite key like (file\_id, last\_changed).

If we use hashes they will be same only when the two contain the same contents. Since we say that file ids must be unique, this means they will match if and only if they are empty. We might relax that in future when we introduce path tokens.

• Is it reasonable to assume hashes won't collide?

The odds of SHA-1 hashes colliding "accidentally" are vanishingly small.

It is possible that a preimage attack against SHA-1 may be discovered in the future. Since we're not proposing in this document to make revision-ids be SHA-1, if SHA-1 was obsoleted then we could rewrite the contents of revisions but would not need to rename revisions. So the impact of such a migration should just be a format upgrade, and a recommendation (but not requirement) to re-sign revisions.

• If we use hashes, should it be the hash of the representation stored for a directory?

In other words, should we pun the representation of the directory with the form used for validation.

If there's some data stored that's not in the hash it's problematic. The hash in no longer (effectively) uniquely identifies the representation.

It is desirable that we have a hash that covers all data, to guard against bugs, transmission errors, or users trying to hand-hack files. Since we need one hash of everything in the tree, perhaps we should also use it for the fingerprint.

Testaments explicitly separate the form used for hashing/signing from the form used for storage. This allows us to change the storage form without breaking existing GPG signatures. The downside is that we need to do work O(tree) to make a testament, and this slows down signing, verifying and generating bundles. It also means that there is some stored data which is not protected by the signature: this data is less important, but corruption of it would still cause problems. We have encountered some specific problems with disagreement between inventories as to the last-change of files, which is currently unsigned. These problems can be introduced by ghosts.

If we hash the representation, there is still a way to support old signatures, assuming that we never discard irreplaceable information. The signature should say what format it applies to (similar to testaments), and we could transform in memory the tree back to that format.

• Is hashing substantially slower than other possible approaches?

We already hash all the plain files. Except in unusual cases, the directory metadata will be substantially smaller: perhaps 200:1 as a rule of thumb.

When building a bzr tree, we spend on the order of 100ms hashing all the source lines to validate them (about 13MB of source).

• Can you calculate one from a directory in the working tree? Without a basis?

This seems possible with either hashes or revision ids.

Using last\_changed means that calculating the fingerprint from a working tree necessarily requires reading the inventory for the basis revision, so that we know when unchanged files were last changed. With hashes we could calculate them using the working tree information alone. It's true that we will often then compare that information to the basis tree (e.g. for simple bzr diff), but we may only have to compare at the top level, and sometimes we're comparing to a different tree. This also

touches on whether we should store <u>last\_modified</u> for files, rather than directories.

For revision ids we need to assign a value to use for uncommitted changes, but see below about the problems of this.

In some ways it would be elegant to say (hypothetical):

```
wt.get_root().get_last_modified() == branch.get_last_revisi
```

to know that nothing was changed; but this may not be much better than

```
wt.get_root().get_hash() ==
    branch.get_basis().get_root().get_hash()
```

 Can you use this to compare (directories from) two working trees?

If you can generate it from a working tree, you should be able to use it to compare them.

This does rule out for example using last\_modified=None or ='current:' to mean "changed in the working tree." Even if this is not supported there seems some risk that we would get the same fingerprint for trees that are actually different.

We could assign a hypothetical revision id to the tree for uncommitted files. In that case there is some risk that the notyet-committed id would become visible or committed.

• Can we use an "approximate basis"?

When using radix trees, you may need context beyond the specific directory being compared.

• Can you get the fingerprint of parents directories with only selected file ids taken from the working tree?

With hashes, we'd want to carry through the unselected files and directories from the values they had in the parent revision.

• Are unbalanced trees a significant problem? Trees can be unbalanced by having many directories (deep or wide), or many files per directory.

For small trees like bzr, 744 of 874 are in the bzrlib subtree. In general, larger trees are more balanced, because humans, editors and other tools have trouble managing very unbalanced trees. But there are exceptions: Aaron has one tree with 20,000 generated but versioned entries in one directory.

- Should we use a radix tree approach where fingerprints are calculated on a synthetic tree that is by definition balanced, even when the actual tree is unbalanced?
- What are the specific advantages of using recursive-lastmodified rather than hashes?

It may be a smaller step change.

It's a bidirectional link: given a directory text identifier (file\_id, last\_changed) you can look up the revision that last changed it.

From the preceding, even without the per-file graph you can skip through the history of this file: go to the last-changed revision, look at all its parents and repeat.

• Is it a smaller change to use recursive-last-modified on directories?

Probably yes:

1. We can just put it into the current inventory format without changing anything else.

By contrast to use a hash we'd have to either split up the inventory as stored, or change the sort order for the inventory, or synthesize per-directory inventories in memory for hashing.

However, xml is somewhat redundant and slow to parse/generate; and reading the whole thing before comparing some sections is only a partial win. It may be a smaller change but we'd be preserving things we want to change.

- 1. At present we rarely hash storage representations, only file texts. This is not a large technical change, but it is a conceptual change. This has some consequences for how we can upgrade it in future: all the changed directories need to be rewritten up to the revision level.
- 1. If we address directories by hash we need hash-addressed storage.
- 1. If we address directories by hash then for consistency we'd probably (not necessarily) want to address file texts by hash.
- 1. The per-file graph can't be indexed by hash because they can converge, so we need to either rework or dispose of the per-file graph.
- Any possibilities for avoiding hashes recurring?
  - 1. Hash along with an identification of the parents (as in hg). Then you can't convert a tree without all its basis trees, and there is still convergence when the same merge is done by two people, and you can't create it directly from the working tree.
  - 1. Include last-modified revision id in the hash.

- 1. Index by (revision, hash) or vice versa.
- 1. Store a per-file graph and allow it to have repeated keys. The graph would tell you about all the parent texts ever seen; you would need to use revision graph information to resolve ambiguities.
- What are the specific disadvantages of using recursive-lastmodified rather than hashes?

To calculate the last-changed revision, given the last-changed information of the contained files, you need to look at the revision graph. They're not enough because you need to know the relations between the mentioned revisions. In a merge it's possible the correct directory last-modified will not be the same as that of any of the files within it. This can also happen when a file is removed (deleted or renamed) from a directory.

• Should we split up storage of the inventories?

This is not quite the same but connected.

• How does this relate to per-file/per-directory hashes?

If the version of a file or directory is identified by a hash, we can't use that to point into a per-file graph. We can have a graph indexed by (file\_id, hash, revision\_id). The last-modified could be stored as part of this graph.

The graph would no longer be core data; it could be always present but might be rebuilt. Treating it as non-core data may make some changes like shallow branches easier?

• How do you ask a tree for a given text?

Right now we say

```
revision_tree.get_file_lines(file_id)
```

so the choice of storage is hidden behind the revision tree: it could be accessed by (file\_id, last\_changed) or by hash or otherwise.

At the moment the Repository exports a friend api to RevisionTree, currently usually talking in VersionedFiles.

We probably wouldn't want Repository to expose a get\_text\_for\_sha1() interface because that would be very difficult to support on old repositories or on foreign branches.

#### 1.4.1.4 Conclusions

- 1.4.1.5 Design changes
- 1.4.1.6 API changes

• Home | Documentation | Developer Document Catalog (2.2b1) » Plans » previous | next

Whome | Documentation | Developer Document Catalog (2.2b1) » Plans » previous | next

# co-located branches

At the moment, each Bazaar branch has a separate directory in the file system. While this works well, and makes it very easy to discover branches there are several situations where it might be useful to also support multiple branches under the same file system directory.

There is an experimental implementation for Bazaar available as a plugin at http://people.samba.org/bzr/jelmer/bzr-local-branches/trunk. This was the original proof-of-concept and doesn't yet use the API documented here.

# Rationale

Allowing multiple branches to live under the same directory in the file system means that it is possible to very easily share the same working tree and repository between those branches, without having a lot of fs infrastructure.

Git and Mercurial (can) store multiple branches under a single directory in the file system - per repository, so to speak. In order for this to be accessible in Bazaar, Bazaar needs to have the right APIs and UI for accessing these branches.

# **Use Cases**

Carla has a large C-based project with a large tree and a lot of .o files that get generated as part of her build process. She doesn't want to create a new working tree for each new branch but simply uses "bzr switch" to switch between the different colocated branches that all use the same working tree.

Brad has a single project with a lot of related branches. He works on them and occasionally pushes all of those branches to a remote host using a single push command.

Joe follows one of his co-workers local branches in Mercurial by pulling into Bazaar.

# Implementation

# **UI** Changes

Bazaar URLs need to have some way to specify a colocated branch other than the current HEAD. Several options have been discussed, each with its own advantages and disadvantages: This was discussed on the mailing list, most notably the use of a ";branch=NAME" suffix as well as a special separation character (+, =, etc), but no final conclusion was reached.

https://lists.ubuntu.com/archives/bazaar/2008q4/050105.html

### Code Changes

BzrDir should support a BzrDir.supports\_colocated\_branches() call as well as BzrDir.colocated\_branches property that contains a colocated branch container, that can be used to add / remove colocated branches as well as change the currently active colocated branch.

```
class ColocatedBranchContainer(object):
    def get_active_branch_name(self):
        """Returns the name of the currently active branch.
        This can be None if no branch is currently active.
        """
    def get_active_branch(self):
        """Returns the currently active branches' Branch obje
    def get_branch(self, name):
        """Returns the Branch object for the specified branch
    def available_branches(self):
        """Returns a set with the names of the available branch
```

```
def set_active_branch(self, name):
    """Set the currently active branch."""
    def destroy_branch(self, name):
        """Destroy the specified branch.
        This will remove the branch from disk."""
```

If the particular BzrDir implementation doesn't support colocated branches, it can just return a dummy container that just contains a HEAD branch.

Looms can of course return a container with all their threads.

BzrDir.find\_branches() should take into account the colocated branches when iterating over its branches.

# Schema Changes

No format changes are necessary at first; at least, even if Bazaar provides the right infrastructure it doesn't have to support this feature in its own file formats.

Eventually, Bazaar could easily support colocated branches by just creating a new branch transport for each colocated branch and have a "regular" branch live there. This would require something like BzrDirMeta2 though.

# Unresolved Issues

- What about colocated looms ?
- What character to use to name colocated branches in URLs?

• Home | Documentation | Developer Document Catalog (2.2b1) » Plans » previous | next

Whome | Documentation | Developer Document Catalog (2.2b1) » Plans » previous | next

# Bazaar Windows Shell Extension Options

- Introduction
- Background Information
  - The facts about shell extensions
  - Analysis of TortoiseSVN code
  - Analysis of existing TortoiseBzr code
- Detailed Implementation Strategy
  - External Command Processor
  - Performance considerations
  - RPC options
  - Vista versus XP
  - Reuse of TSVNCache?
  - Reuse of this code by Mercurial or other Python based VCS systems?
- Implementation plan
- Alternative Implementation Strategies
  - Implement Completely in Python

# Introduction

This document details the implementation strategy chosen for the Bazaar Windows Shell Extensions, otherwise known as TortoiseBzr, or TBZR. As justification for the strategy, it also describes the general architecture of Windows Shell Extensions, then looks at the C++ implemented TortoiseSvn and the Python implemented TortoiseBzr, and discusses alternative implementation strategies, and the reasons they were not chosen.

The following points summarize the strategy:

- Main shell extension code will be implemented in C++, and be as thin as possible. It will not directly do any VCS work, but instead will perform all operations via either external applications or an RPC server.
- Most VCS operations will be performed by external applications. For example, committing changes or viewing history will spawn a child process that provides its own UI.
- For operations where spawning a child process is not practical, an external RPC server will be implemented in Python and will directly use the VCS library. In the short term, there will be no attempt to create a general purpose RPC mechanism, but instead will be focused on keeping the C++ RPC client as thin, fast and dumb as possible.

# **Background Information**

### The facts about shell extensions

Well - the facts as I understand them :)

Shell Extensions are COM objects. They are implemented as DLLs which are loaded by the Windows shell. There is no facility for shell extensions to exist in a separate process - DLLs are the only option, and they are loaded into other processes which take advantage of the Windows shell (although obviously this DLL is free to do whatever it likes).

For the sake of this discussion, there are 2 categories of shell extensions:

- Ones that create a new "namespace". The file-system itself is an example of such a namespace, as is the "Recycle Bin". For a user-created example, picture a new tree under "My Computer" which allows you to browse a remote server it creates a new, stand-alone tree that doesn't really interact with the existing namespaces.
- Ones that enhance existing namespaces, including the filesystem. An example would be an extension which uses Icon Overlays to modify how existing files on disk are displayed or add items to their context menu, for example.

The latter category is the kind of shell extension relevant for TortoiseBzr, and it has an important implication - it will be pulled into any process which uses the shell to display a list of files. While this is somewhat obvious for Windows Explorer (which many people consider the shell), every other process that shows a FileOpen/FileSave dialog will have these shell extensions loaded into its process space. This may surprise many people - the simple fact of allowing the user to select a filename will result in an unknown number of DLLs being loaded into your process. For a concrete example, when notepad.exe first starts with an empty file it is using around 3.5MB of RAM. As soon as the FileOpen dialog is loaded, TortoiseSvn loads well over 20 additional DLLs, including the MSVC8 runtime, into the Notepad process causing its memory usage (as reported by task manager) to more than double - all without doing anything tortoise specific at all. (In fairness, this illustration is contrived - the code from these DLLs are already in memory and there is no reason to suggest TSVN adds any other unreasonable burden - but the general point remains valid.)

This has wide-ranging implications. It means that such shell extensions should be developed using a tool which can never cause conflict with arbitrary processes. For this very reason, MS recommend against using .NET to write shell extensions[1], as there is a significant risk of being loaded into a process that uses a different version of the .NET runtime, and this will kill the process. Similarly, Python implemented shell extension may well conflict badly with other Python implemented applications (and will certainly kill them in some situations). A similar issue exists with GUI toolkits used - using (say) PyGTK directly in the shell extension would need to be avoided (which it currently is best I can tell). It should also be obvious that the shell extension will be in many processes simultaneously, meaning use of a simple log-file (for example) is problematic.

In practice, there is only 1 truly safe option - a low-level language (such as C/C++) which makes use of only the win32 API, and a static version of the C runtime library if necessary. Obviously, this sucks from our POV. :)

[1]:

http://blogs.msdn.com/oldnewthing/archive/2006/12/18/1317290.aspx

# Analysis of TortoiseSVN code

TortoiseSVN is implemented in C++. It relies on an external process to perform most UI (such as diff, log, commit etc.) commands, but it appears to directly embed the SVN C libraries for the purposes of obtaining status for icon overlays, context menu, drag&drop, etc.

The use of an external process to perform commands is fairly simplistic in terms of parent and modal windows. For example, when selecting "Commit", a new process starts and *usually* ends up as the foreground window, but it may occasionally be lost underneath the window which created it, and the user may accidently start many processes when they only need 1. Best I can tell, this isn't necessarily a limitation of the approach, just the implementation.

Advantages of using the external process is that it keeps all the UI code outside Windows explorer - only the minimum needed to perform operations directly needed by the shell are part of the "shell extension" and the rest of TortoiseSvn is "just" a fairly large GUI application implementing many commands. The command-line to the app has even been documented for people who wish to automate tasks using that GUI. This GUI is also implemented in C++ using Windows resource files.

TortoiseSvn has an option (enabled by default) which enabled a cache using a separate process, aptly named TSVNCache.exe. It uses a named pipe to accept connections from other processes for various operations. When enabled, TSVN fetches most (all?) status information from this process, but it also has the option to talk directly to the VCS, along with options to disable functionality in various cases.

There doesn't seem to be a good story for logging or debugging which is what you expect from C++ based apps. :( Most of the heavy lifting is done by the external application, which might offer better facilities.

### Analysis of existing TortoiseBzr code

The existing code is actually quite cool given its history (SoC student, etc), so this should not be taken as criticism of the implementer nor of the implementation. Indeed, many criticisms are also true of the TortoiseSvn implementation - see above. However, I have attempted to list the bad things rather than the good things so a clear future strategy can be agreed, with all limitations understood.

The existing TortoiseBzr code has been ported into Python from other tortoise implementations (probably svn). This means it is very nice to implement and develop, but suffers the problems described above - it is likely to conflict with other Python based processes, and it means the entire CPython runtime and libraries are pulled into many arbitrary processes.

The existing TortoiseBzr code pulls in the bzrlib library to determine the path of the bzr library, and also to determine the status of files, but uses an external process for most GUI commands - ie, very similar to TortoiseSvn as described above - and as such, all comments above apply equally here - but note that the bzr library *is* pulled into the shell, and therefore every application using the shell. The GUI in the external application is written in PyGTK, which may not offer the best Windows "look and feel", but that discussion is beyond the scope of this document.

It has a better story for logging and debugging for the developer - but not for diagnosing issues in the field - although again, much of the heavy lifting remains done by the external application.

It uses a rudimentary in-memory cache for the status of files and directories, the implementation of which isn't really suitable (ie, no theoretical upper bound on cache size), and also means that there is no sharing of cached information between processes, which is unfortunate (eg, imagine a user using Windows explorer, then switching back to their editor) and also error prone (it's possible the editor will check the file in, meaning Windows explorer will be showing stale data). This may be possible to address via file-system notifications, but a shared cache would be preferred (although clearly more difficult to implement).

One tortoise port recently announced a technique for all tortoise ports to share the same icon overlays to help work around a limitation in Windows on the total number of overlays (it's limited to 15, due to the number of bits reserved in a 32bit int for overlays). TBZR needs to take advantage of that (but to be fair, this overlay sharing technique was probably done after the TBZR implementation).

The current code appears to recursively walk a tree to check if *any* file in the tree has changed, so it can reflect this in the parent directory status. This is almost certainly an evil thing to do (Shell Extensions are optimized so that a folder doesn't even need to look in its direct children for another folder, let alone recurse for any reason at all. It may be a network mounted drive that doesn't perform at all.)

Although somewhat dependent on bzr itself, we need a strategy for binary releases (ie, it assumes python.exe, etc) and integration into an existing "blessed" installer.

Trivially, the code is not PEP8 compliant and was written by someone fairly inexperienced with the language.

# **Detailed Implementation Strategy**

We will create a hybrid Python and C++ implementation. In this model, we would still use something like TSVNCache.exe (this external process doesn't have the same restrictions as the shell extension itself) but go one step further - use this remote process for *all* interactions with bzr, including status and other "must be fast" operations. This would allow the shell extension itself to be implemented in C++, but still take advantage of Python for much of the logic.

A pragmatic implementation strategy will be used to work towards the above infrastructure - we will keep the shell extension implemented in Python - but without using bzrlib. This allows us to focus on this shared-cache/remote-process infrastructure without immediately re-implementing a shell extension in C++. Longer term, once the infrastructure is in place and as optimized as possible, we can move to C++ code in the shell calling our remote Python process. This port should try and share as much code as possible from TortoiseSvn, including overlay handlers.

### **External Command Processor**

The external command application (ie, the app invoked by the shell extension to perform commands) can remain as-is, and remain a "shell" for other external commands. The implementation of this application is not particularly relevant to the shell extension, just the interface to the application (ie, its command-line) is. In the short term this will remain PyGTK and will only change if there is compelling reason - cross-platform GUI tools are a better for bazaar than Windows specific ones, although native look-and-feel is important. Either way, this can change independently from the shell extension.

# Performance considerations

As discussed above, the model used by Tortoise is that most "interesting" things are done by external applications. Most Tortoise implementations show read-only columns in the "detail" view, and shows a few read only properties in the "Properties" dialog - but most of these properties are "state" related (eg, revision number), or editing of others is done by launching an external application. This means that the shell extension itself really has 2 basic requirements WRT RPC: 1) get the local state of a file and 2) get some named state-related "properties" for a file. Everything else can be built on that.

There are 2 aspects of the shell integration which are performance critical - the "icon overlays" and "column providers".

The short-story with Icon Overlays is that we need to register 12 global "overlay providers" - one for each state we show. Each provider is called for every icon ever shown in Windows explorer or in any application's FileOpen dialog. While most versions of Windows update icons in the background, we still need to perform well. On the positive side, this just needs the simple "local state" of a file - information that can probably be carried in a single byte. On the negative side, it is the shell which makes a synchronous call to us with a single filename as an arg, which makes it difficult to "batch" multiple status requests into a single RPC call.

The story with columns is messier - these have changed significantly for Vista and the new system may not work with the VCS model (see below). However, if we implement this, it will be fairly critical to have high-performance name/value pairs implemented, as described above.

Note that the nature of the shell implementation means we will have a large number of "unrelated" handlers, each called somewhat independently by the shell, often for information about the same file (eg, imagine each of our overlay providers all called in turn with the same filename, followed by our column providers called in turn with the same filename. However, that isn't exactly what happens!). This means we will need a kind of cache, geared towards reducing the number of status or property requests we make to the RPC server.

We will also allow all of the above to be disabled via user preferences. Thus, Icon Overlays could be disabled if it did cause a problem for some people, for example.

### **RPC** options

Due to the high number of calls for icon overlays, the RPC overhead must be kept as low as possible. Due to the client side being implemented in C++, reducing complexity is also a goal. Our requirements are quite simple and no existing RPC options exist we can leverage. It does not seen prudent to build an XMLRPC solution for tbzr - which is not to preclude the use of such a server in the future, but tbzr need not become the "pilot" project for an XMLRPC server given these constraints.

I propose that a custom RPC mechanism, built initially using windows-specific named-pipes, be used. A binary format, designed with an eye towards implementation speed and C++ simplicity, will be used. If we succeed here, we can build on that infrastructure, and even replace it should other more general frameworks materialize.

FWIW, with a Python process at each end, my P4 2.4G machine can achieve around 25000 "calls" per-second across an open named pipe. C++ at one end should increase this a little, but obviously any real work done by the Python side of the process will be the bottleneck. However, this throughput would appear sufficient to implement a prototype.

### Vista versus XP

Let's try and avoid an OS advocacy debate :) But it is probably true that TBZR will, over its life, be used by more Vista computers than XP ones. In short, Vista has changed a number of shell related interfaces, and while TSVN is slowly catching up (http://tortoisesvn.net/vistaproblems) they are a pain.

XP has IColumnProvider (as implemented by Tortoise), but Vista changes this model. The new model is based around "file types" (eg, .jpg files) and it appears each file type can only have 1 provider! TSVN also seems to think the Vista model isn't going to work (see previous link). It's not clear how much effort we should expend on a column system that has already been abandoned by MS. I would argue we spend effort on other parts of the system (ie, the external GUI apps themselves, etc) and see if a path forward does emerge for Vista. We can re-evaluate this based on user feedback and more information about features of the Vista property system.

### Reuse of TSVNCache?

The RPC mechanism and the tasks performed by the RPC server (rpc, file system crawling and watching, device notifications, caching) are very similar to those already implemented for TSVN and analysis of that code shows that it is not particularly tied to any VCS model. As a result, consideration should be given to making the best use of this existing debugged and optimized technology.

Discussions with the TSVN developers have indicated that they would prefer us to fork their code rather than introduce complexity and instability into their code by attempting to share it. See the follow-ups to http://thread.gmane.org/gmane.comp.version-control.subversion.tortoisesvn.devel/32635/focus=32651 for details.

For background, the TSVNCache process is fairly sophisticated - but mainly in areas not related to source control. It has had various performance tweaks and is smart in terms of minimizing its use of resources when possible. The 'cloc' utility counts ~5000 lines of C++ code and weighs in just under 200KB on disk (not including headers), so this is not a trivial application. However, the code that is of most interest (the crawlers, watchers and cache) are roughly ~2500 lines of C++. Most of the source files only depend lightly on SVN specifics, so it would not be a huge job to make the existing code talk to Bazaar. The code is thread-safe, but not particularly thread-friendly (ie, fairly coarse-grained locks are taken in most cases).

In practice, this give us 2 options - "fork" or "port":

• Fork the existing C++ code, replacing the existing source-control code with code that talks to Bazaar. This would involve introducing a Python layer, but only at the layers where we need to talk to bzrlib. The bulk of the code would remain in C++.

This would have the following benefits:

- May offer significant performance advantages in some cases (eg, a cache-hit would never enter Python at all.)
- Quickest time to a prototype working the existing working code can be used quicker.

And the following drawbacks:

- More complex to develop. People wishing to hack on it must be on Windows, know C++ and own the most recent MSVC8.
- More complex to build and package: people making binaries must be on Windows and have the most recent MSVC8.
- $\circ\,$  Is tied to Windows it would be impractical for this to be
cross-platform, even just for test purposes (although parts of it obviously could).

• Port the existing C++ code to Python. We would do this almost "line-for-line", and attempt to keep many optimizations in place (or at least document what the optimizations were for ones we consider dubious). For the windows versions, pywin32 and ctypes would be leaned on - there would be no C++ at all.

This would have the following benefits:

- Only need Python and Python skills to hack on it.
- No C++ compiler needed means easier to cut releases
- Python makes it easier to understand and maintain it should appear much less complex than the C++ version.

And the following drawbacks:

- Will be slower in some cases eg, a cache-hit will involve executing Python code.
- Will take longer to get a minimal system working. In practice this probably means the initial versions will not be as sophisticated.

Given the above, there are two issues which prevent Python being the clear winner: (1) will it perform OK? (2) How much longer to a prototype?

My gut feeling on (1) is that it will perform fine, given a suitable Python implementation. For example, Python code that simply looked up a dictionary would be fast enough - so it all depends on how fast we can make our cache. Re (2), it should be possible to have a "stub" process (did almost nothing in terms of caching or crawling, but could be connected to by the shell) in a 8 hours, and some crawling and caching in 40. Note that this is separate from the work included for the shell extension itself (the implementation of which is largely independent of the TBZRCache implementation). So given the lack of a deadline for any particular feature and the better long-term fit of using Python, the conclusion is that we should "port" TSVN for bazaar.

# Reuse of this code by Mercurial or other Python based VCS systems?

Incidentally, the hope is that this work can be picked up by the Mercurial project (or anyone else who thinks it is of use). However, we will limit ourselves to attempting to find a clean abstraction for the parts that talk to the VCS (as good design would dictate regardless) and then try and assist other projects in providing patches which work for both of us. In other words, supporting multiple VCS systems is not an explicit goal at this stage, but we would hope it is possible in the future.

# Implementation plan

The following is a high-level set of milestones for the implementation:

- Design the RPC mechanism used for icon overlays (ie, binary format used for communication).
- Create Python prototype of the C++ "shim": modify the existing TBZR Python code so that all references to "bzrlib" are removed. Implement the client side of the RPC mechanism and implement icon overlays using this RPC mechanism.
- Create initial implementation of RPC server in Python. This will use bzrlib, but will also maintain a local cache to achieve the required performance. File crawling and watching will not be implemented at this stage, but caching will (although cache persistence might be skipped).
- Analyze performance of prototype. Verify that technique is feasible and will offer reasonable performance and user experience.
- Implement file watching, crawling etc by "porting" TSVNCache code to Python, as described above.
- Implement C++ shim: replace the Python prototype with a lightweight C++ version. We will fork the current TSVN sources, including its new support for sharing icon overlays (although advice on how to setup this fork is needed!)
- Implement property pages and context menus in C++. Expand RPC server as necessary.
- Create binary for alpha releases, then go round-and-round until its baked.

# Alternative Implementation Strategies

Only one credible alternative strategy was identified, as discussed below. No languages other than Python and C++ were considered; Python as the bzr library and existing extensions are written in Python and otherwise only C++ for reasons outlined in the background on shell extensions above.

#### **Implement Completely in Python**

This would keep the basic structure of the existing TBZR code, with the shell extension continuing to pull in Python and all libraries used by Bzr into various processes.

Although implementation simplicity is a key benefit to this option, it was not chosen for various reasons, e.g. the use of Python means that there is a larger chance of conflicting with existing applications, or even existing Python implemented shell extensions. It will also increase the memory usage of all applications which use the shell. While this may create problems for a small number of users, it may create a wider perception of instability or resource hogging.

Image by the previous of the p

Whome | Documentation | Developer Document Catalog (2.2b1) » Plans » previous | next

# CHK Optimized index

Our current btree style index is nice as a general index, but it is not optimal for Content-Hash-Key based content. With CHK, the keys themselves are hashes, which means they are randomly distributed (similar keys do not refer to similar content), and they do not compress well. However, we can create an index which takes advantage of these abilites, rather than suffering from them. Even further, there are specific advantages provided by groupcompress, because of how individual items are clustered together.

Btree indexes also rely on zlib compression, in order to get their compact size, and further has to try hard to fit things into a compressed 4k page. When the key is a sha1 hash, we would not expect to get better than 20bytes per key, which is the same size as the binary representation of the hash. This means we could write an index format that gets approximately the same on-disk size, without having the overhead of zlib.decompress. Some thought would still need to be put into how to efficiently access these records from remote.

# **Required information**

For a given groupcompress record, we need to know the offset and length of the compressed group in the .pack file, and the start and end of the content inside the uncompressed group. The absolute minimum is slightly less, but this is a good starting point. The other thing to consider, is that for 1M revisions and 1M files, we'll probably have 10-20M CHK pages, so we want to make sure we have an index that can scale up efficiently.

- 1. A compressed sha hash is 20-bytes
- Pack files can be > 4GB, we could use an 8-byte (64-bit) pointer, or we could store a 5-byte pointer for a cap at 1TB. 8bytes still seems like overkill, even if it is the natural next size up.
- 3. An individual group would never be longer than 2^32, but they will often be bigger than 2^16. 3 bytes for length (16MB) would be the minimum safe length, and may not be safe if we expand groups for large content (like ISOs). So probably 4-bytes for group length is necessary.
- 4. A given start offset has to fit in the group, so another 4-bytes.
- 5. Uncompressed length of record is based on original size, so 4bytes is expected as well.
- 6. That leaves us with 20+8+4+4+4 = 40 bytes per record. At the moment, btree compression gives us closer to 38.5 bytes per record. We don't have perfect compression, but we also don't have >4GB pack files (and if we did, the first 4GB are all under then 2^32 barrier :).

If we wanted to go back to the "minimal" amount of data that we would need to store.

1. 8 bytes of a sha hash are generally going to be more than enough to fully determine the entry (see Partial hash). We could

support some amount of collision in an index record, in exchange for resolving it inside the content. At least in theory, we don't *have* to record the whole 20-bytes for the sha1 hash. (8-bytes gives us less than 1 in 1000 chance of a single collision for 10M nodes in an index)

- 2. We could record the start and length of each group in a separate location, and then have each record reference the group by an 'offset'. This is because we expect to have many records in the same group (something like 10k or so, though we've fit >64k under some circumstances). At a minimum, we have one record per group so we have to store at least one reference anyway. So the maximum overhead is just the size and cost of the dereference (and normally will be much much better than that.)
- 3. If a group reference is an 8-byte start, and a 4-byte length, and we have 10M keys, but get at least 1k records per group, then we would have 10k groups. So we would need 120kB to record all the group offsets, and then each individual record would only need a 2-byte group number, rather than a 12-byte reference. We could be safe with a 4-byte group number, but if each group is ~1MB, 64k groups is 64GB. We can start with 2-byte, but leave room in the header info to indicate if we have more than 64k group entries. Also, current grouping creates groups of 4MB each, which would make it 256GB, to create 64k groups. And our current chk pages compress down to less than 100 bytes each (average is closer to 40 bytes), which for 256GB of raw data, would amount to 2.7 billion CHK records. (This will change if we start to use CHK for text records, as they do not compress down as small.) Using 100 bytes per 10M chk records, we have 1GB of compressed chk data, split into 4MB groups or 250 total groups. Still << 64k groups. Conversions could create 1 chk record at a time, creating a group for each, but they would be foolish to not commit a write group after 10k revisions (assuming

6 CHK pages each).

4. We want to know the start-and-length of a record in the decompressed stream. This could actually be moved into a miniindex inside the group itself. Initial testing showed that storing an expanded "key => start,offset" consumed a considerable amount of compressed space. (about 30% of final size was just these internal indices.) However, we could move to a pure "record 1 is at location 10-20", and then our external index would just have a single 'group entry number'.

There are other internal forces that would give a natural cap of 64k entries per group. So without much loss of generality, we could probably get away with a 2-byte 'group entry' number. (which then generates an 8-byte offset + endpoint as a header in the group itself.)

- 5. So for 1M keys, an ideal chk+group index would be:
  - a. 6-byte hash prefix
  - b. 2-byte group number
  - c. 2-byte entry in group number
  - d. a separate lookup of 12-byte group number to offset + length
  - e. a variable width mini-index that splits X bits of the key. (to maintain small keys, low chance of collision, this is *not* redundant with the value stored in (a)) This should then dereference into a location in the index. This should probably be a 4-byte reference. It is unlikely, but possible, to have an index >16MB. With an 10-byte entry, it only takes 1.6M chk nodes to do so. At the smallest end, this will probably be a 256-way (8-bits) fan out, at the high end it could go up to 64k-way (16-bits) or maybe even 1M-way (20-bits). (64k-way should handle up to 5-16M nodes and still allow a cheap <4k

read to find the final entry.)

So the max size for the optimal groupcompress+chk index with 10M entries would be:

So 101MiB which breaks down as 100MiB for the actual entries, 0.75MiB for the group records, and 0.25MiB for the mini index.

- 1. Looking up a key would involve:
  - a. Read xx bytes to get the header, and various config for the index. Such as length of the group records, length of mini index, etc.
  - b. Find the offset in the mini index for the first YY bits of the key. Read the 4 byte pointer stored at that location (which may already be in the first content if we pre-read a minimum size.)
  - c. Jump to the location indicated, and read enough bytes to find the correct 12-byte record. The mini-index only indicates the start of records that start with the given prefix. A 64k-way index resolves 10MB records down to 160 possibilities. So at 12 bytes each, to read all would cost 1920 bytes to be read.
  - Determine the offset for the group entry, which is the known start of groups location + 12B\*offset number. Read its 12byte record.
  - e. Switch to the .pack file, and read the group header to determine where in the stream the given record exists. At this point, you have enough information to read the entire

group block. For local ops, you could only read enough to get the header, and then only read enough to decompress just the content you want to get at.

Using an offset, you also don't need to decode the entire group header. If we assume that things are stored in fixedsize records, you can jump to exactly the entry that you care about, and read its 8-byte (start,length in uncompressed) info. If we wanted more redundancy we could store the 20-byte hash, but the content can verify itself.

f. If the size of these mini headers becomes critical (8 bytes per record is 8% overhead for 100 byte records), we could also compress this mini header. Changing the number of bytes per entry is unlikely to be efficient, because groups standardize on 4MiB wide, which is >>64KiB for a 2-byte offset, 3-bytes would be enough as long as we never store an ISO as a single entry in the content. Variable width also isn't a big win, since base-128 hits 4-bytes at just 2MiB.

For minimum size without compression, we could only store the 4-byte length of each node. Then to compute the offset, you have to sum all previous nodes. We require <64k nodes in a group, so it is up to 256KiB for this header, but we would lose partial reads. This should still be cheap in compiled code (needs tests, as you can't do partial info), and would also have the advantage that fixed width would be highly compressible itself. (Most nodes are going to have a length that fits 1-2 bytes.)

An alternative form would be to use the base-128 encoding. (If the MSB is set, then the next byte needs to be added to the current value shifted by 7\*n bits.) This encodes 4GiB in 5 bytes, but stores 127B in 1 byte, and 2MiB in 3 bytes. If we only stored 64k entries in a 4 MiB group, the average size can only be 64B, which fits in a single byte length, so 64KiB for this header, or only 1.5% overhead. We also don't have to compute the offset of *all* nodes, just the ones before the one we want, which is the similar to what we have to do to get the actual content out.

# Partial Hash

The size of the index is dominated by the individual entries (the 1M records). Saving 1 byte there saves 1MB overall, which is the same as the group entries and mini index combined. If we can change the index so that it can handle collisions gracefully (have multiple records for a given collision), then we can shrink the number of bytes we need overall. Also, if we aren't going to put the full 20-bytes into the index, then some form of graceful handling of collisions is recommended anyway.

The current structure does this just fine, in that the mini-index dereferences you to a "list" of records that start with that prefix. It is assumed that those would be sorted, but we could easily have multiple records. To resolve the exact record, you can read both records, and compute the sha1 to decide between them. This has performance implications, as you are now decoding 2x the records to get at one.

The chance of **n** texts colliding with a hash space of **H** is generally given as:

1 - e ^(-n^2 / 2 H)

Or if you use  $H = 2^h$ , where h is the number of bits:

1 - e ^(-n^2 / 2^(h+1))

For 1M keys and 4-bytes (32-bit), the chance of collision is for all intents and purposes 100%. Rewriting the equation to give the number of bits (h) needed versus the number of entries (n) and the desired collision rate (epsilon):

 $h = \log_2(-n^2 / \ln(1-epsilon)) - 1$ 

The denominator ln(1-epsilon) == -epsilon for small values (even @0.1 == -0.105, and we are assuming we want a much lower chance of collision than 10%). So we have:

 $h = \log_2(n^2/epsilon) - 1 = 2 \log_2(n) - \log_2(epsilon) - 1$ 

Given that epsilon will often be very small and n very large, it can be more convenient to transform it into epsilon =  $10^{-E}$  and n =  $10^{N}$ , which gives us:

```
h = 2 * log_2(10^N) - 2 log_2(10^-E) - 1
h = log_2(10) (2N + E) - 1
h ~ 3.3 (2N + E) - 1
```

Or if we use number of bytes h = 8H:

 $H \sim 0.4 (2N + E)$ 

This actually has some nice understanding to be had. For every order of magnitude we want to increase the number of keys (at the same chance of collision), we need  $\sim$ 1 byte (0.8), for every two orders of magnitude we want to reduce the chance of collision we need the same extra bytes. So with 8 bytes, you can have 20 orders of magnitude to work with, 10^10 keys, with guaranteed collision, or 10 keys with 10^-20 chance of collision.

Putting this in a different form, we could make epsilon == 1/n. This gives us an interesting simplified form:

 $h = \log_2(n^3) - 1 = 3 \log_2(n) - 1$ 

writing n as 10^N, and H=8h:

h = 3 N log\_2(10) - 1 =~ 10 N - 1 H ~ 1.25 N So to have a one in a million chance of collision using 1 million keys, you need ~59 bits, or slightly more than 7 bytes. For 10 million keys and a one in 10 million chance of any of them colliding, you can use 9 (8.6) bytes. With 10 bytes, we have a one in a 100M chance of getting a collision in 100M keys (substituting back, the original equation says the chance of collision is 4e-9 for 100M keys when using 10 bytes.)

Given that the only cost for a collision is reading a second page and ensuring the sha hash actually matches we could actually use a fairly "high" collision rate. A chance of 1 in 1000 that you will collide in an index with 1M keys is certainly acceptible. (note that isn't 1 in 1000 of those keys will be a collision, but 1 in 1000 that you will have a *single* collision). Using a collision chance of 10^-3, and number of keys 10^6, means we need (12+3)\*0.4 = 6 bytes. For 10M keys, you need (14+3)\*0.4 = 6.8 aka 7. We get that extra byte from the miniindex. In an index with a lot of keys, you want a bigger fan-out up front anyway, which gives you more bytes consumed and extends your effective key width.

Also taking one more look at  $H \sim 0.4$  (2N + E), you can rearrange and consider that for every order of magnitude more keys you insert, your chance for collision goes up by 2 orders of magnitude. But for 100M keys, 8 bytes gives you a 1 in 10,000 chance of collision, and that is gotten at a 16-bit fan-out (64k-way), but for 100M keys, we would likely want at least 20-bit fan out.

You can also see this from the original equation with a bit of rearranging:

epsilon = 1 - e^(-n^2 / 2^(h+1))
epsilon = 1 - e^(-(2^N)^2 / (2^(h+1))) = 1 - e^(-(2^(2N))(2^-(h = 1 - e^(-(2^(2N - h - 1))))

Such that you want 2N - h to be a very negative integer, such that

 $2^{-x}$  is thus very close to zero, and  $1-e^{-0} = 0$ . But you can see that if you want to double the number of source texts, you need to quadruple the number of bits.

# Scaling Sizes

### Scaling up

We have said we want to be able to scale to a tree with 1M files and 1M commits. With a 255-way fan out for chk pages, you need 2 internal nodes, and a leaf node with 16 items. (You maintain 2 internal nodes up until 16.5M nodes, when you get another internal node, and your leaf nodes shrink down to 1 again.) If we assume every commit averages 10 changes (large, but possible, especially with large merges), then you get 1 root + 10\*(1 internal + 1 leaf node) per commit or 21 nodes per commit. At 1M revisions, that is 21M chk nodes. So to support the 1Mx1M project, we really need to consider having up to 100M chk nodes.

Even if you went up to 16M tree nodes, that only bumps us up to 31M chk nodes. Though it also scales by number of changes, so if you had a huge churn, and had 100 changes per commit and a 16M node tree, you would have 301M chk nodes. Note that 8 bytes (64-bits) in the prefix still only gives us a 0.27% chance of collision (1 in 370). Or if you had 370 projects of that size, with all different content, *one* of them would have a collision in the index.

We also should consider that you have the (parent\_id, basename) => file\_id map that takes up its own set of chk pages, but testing seems to indicate that it is only about 1/10th that of the id\_to\_entry map. (rename,add,delete are much less common then content changes.)

As a point of reference, one of the largest projects today OOo, has only 170k revisions, and something less than 100k files (and probably 4-5 changes per commit, but their history has very few merges, being a conversion from CVS). At 100k files, they are probably just starting to hit 2-internal nodes, so they would end up with 10 pages per commit (as a fair-but-high estimate), and at 170k revs, that would be 1.7M chk nodes.

#### Scaling down

While it is nice to scale to a 16M files tree with 1M files (100M total changes), it is also important to scale efficiently to more *real world* scenarios. Most projects will fall into the 255-64k file range, which is where you have one internal node and 255 leaf nodes (1-2 chk nodes per commit). And a modest number of changes (10 is generally a high figure). At 50k revisions, that would give you 50\*2\*10=500k chk nodes. (Note that all of python has 303k chk nodes, all of launchpad has 350k, mysql-5.1 in gc255 rather than gc255big had 650k chk nodes, [depth=3].)

So for these trees, scaling to 1M nodes is more than sufficient, and allows us to use a 6-byte prefix per record. At a minimum, group records could use a 4-byte start and 3-byte length, but honestly, they are a tiny fraction of the overall index size, and it isn't really worth the implementation cost of being flexible here. We can keep a field in the header for the group record layout (8, 4) and for now just assert that this size is fixed.

# Other discussion

#### group encoding

In the above scheme we store the group locations as an 8-byte start, and 4-byte length. We could theoretically just store a 4-byte length, and then you have to read all of the groups and add them up to determine the actual start position. The trade off is a direct jump-to-location versus storing 3x the data. Given when you have 64k groups you will need only .75MiB to store it, versus the 120MB for the actual entries, this seems to be no real overhead. Especially when you consider that 10M chk nodes should fit in only 250 groups, so total data is actually only 3KiB. Then again, if it was only 1KiB it is obvious that you would read the whole thing in one pass. But again, see the pathological "conversion creating 1 group per chk page" issue.

Also, we might want to support more than 64k groups in a given index when we get to the point of storing file content in a CHK index. A lot of the analysis about the number of groups is based on the 100 byte compression of CHK nodes, which would not be true with filecontent. We should compress well, I don't expect us to compress that well. Launchpad shows that the average size of a content record is about 500-600 bytes (after you filter out the ~140k that are NULL content records). At that size, you expect to get approx 7k records per group, down from 40k. Going further, though, you also want to split groups earlier, since you end up with better compression. so with 100,000 unique file texts, you end up with ~100 groups. With 1M revisions @ 10 changes each, you have 10M file texts, and would end up at 10,485 groups. That seems like more 64k groups is still more than enough head room. You need to fit only 100 entries per group, to get down to where you are getting into trouble (and have 10M file texts.) Something to keep an eye on, but unlikely to be

something that is strictly a problem.

Still reasonable to have a record in the header indicating that index entries use a 2-byte group entry pointer, and allow it to scale to 3 (we may also find a win scaling it down to 1 in the common cases of <250 groups). Note that if you have the full 4MB groups, it takes 256 GB of compressed content to fill 64k records. And our groups are currently scaled that we require at least 1-2MB before they can be considered 'full'.

#### variable length index entries

The above had us store 8-bytes of sha hash, 2 bytes of group number, and 2 bytes for record-in-group. However, since we have the variable-pointer mini-index, we could consider having those values be 'variable length'. So when you read the bytes between the previous-and-next record, you have a parser that can handle variable width. The main problem is that to encode start/stop of record takes some bytes, and at 12-bytes for a record, you don't have a lot of space to waste for a "end-of-entry" indicator. The easiest would be to store things in base-128 (high bit indicates the next byte also should be included).

#### storing uncompressed offset + length

To get the smallest index possible, we store only a 2-byte 'record indicator' inside the index, and then assume that it can be decoded once we've read the actual group. This is certainly possible, but it represents yet another layer of indirection before you can actually get content. If we went with variable-length index entries, we could probably get most of the benefit with a variable-width start-of-entry value. The length-of-content is already being stored as a base128 integer starting at the second byte of the uncompressed data (the first being the record type, fulltext/delta). It complicates some of our other processing, since we would then only know how much to decompress to get the start of the record.

Another intriguing possibility would be to store the *end* of the record in the index, and then in the data stream store the length and type information at the *end* of the record, rather than at the beginning (or possibly at both ends). Storing it at the end is a bit unintuitive when you think about reading in the data as a stream, and figuring out information (you have to read to the end, then seek back) But a given GC block does store the length-of-uncompressed-content, which means we can trivially decompress, jump to the end, and then walk-backwards for everything else.

Given that every byte in an index entry costs 10MiB in a 10M index, it is worth considering. At 4MiB for a block, base 128 takes 4 bytes to encode the last 50% of records (those beyond 2MiB), 3 bytes for everything from 16KiB => 2MiB. So the expected size is for all intents and purposes, 3.5 bytes. (Just due to an unfortunate effect of where the boundary is that you need more bytes.) If we capped the data at 2MB, the expected drops to just under 3 bytes. Note that a flat 3bytes could decode up to 16MiB, which would be much better for our purpose, but wouldn't let us write groups that had a record after 16MiB, which doesn't work for the ISO case. Though it works *absolutely* fine for the CHK inventory cases (what we have today).

#### null content

At the moment, we have a lot of records in our per-file graph that refers to empty content. We get one for every symlink and directory, for every time that they change. This isn't specifically relevant for CHK pages, but for efficiency we could certainly consider setting "group = 0 entry = 0" to mean that this is actually a no-content entry. It means the group block itself doesn't have to hold a record for it, etc. Alternatively we could use "group=FFFF entry = FFFF" to mean the same thing.

#### VF.keys()

At the moment, some apis expect that you can list the references by reading all of the index. We would like to get away from this anyway, as it doesn't scale particularly well. However, with this format, we no longer store the exact value for the content. The content is self describing, and we *would* be storing enough to uniquely decide which node to read. Though that is actually contained in just 4-bytes (2-byte group, 2-byte group entry).

We use VF.keys() during 'pack' and 'autopack' to avoid asking for content we don't have, and to put a counter on the progress bar. For the latter, we can just use index.key\_count() for the former, we could just properly handle AbsentContentFactory.

#### More than 64k groups

Doing a streaming conversion all at once is still something to consider. As it would default to creating all chk pages in separate groups (300-400k easily). However, just making the number of group block entries variable, and allowing the pointer in each entry to be variable should suffice. At 3 bytes for the group pointer, we can refer to 16.7M groups. It does add complexity, but it is likely necessary to allow for arbitrary cases.

 $\clubsuit$  Home | Documentation | Developer Document Catalog (2.2b1) » Plans » previous | next

In the second se

# Specifications

- Revision Properties An application can set arbitrary perrevision key/value pairs to store app-specific data.
- API versioning bzrlib API versioning.
- Apport error reporting Capture data to report bugs.
- Authentication ring Configuring authentication.
- Bundles All about bzr bundles.
- Container format Notes on a container format for streaming and storing Bazaar data.
- Groupcompress Notes on the compression technology used in CHK repositories.
- Indices The index facilities available within bzrlib.
- Inventories Tree shape abstraction.
- LCA merge A nice new merge algorithm.
- Network protocol Custom network protocol.
- Plugin APIs APIs plugins should use.
- Repositories What repositories do and are used for.
- Repository stream Notes on streaming data for repositories (a layer above the container format).
- Bazaar and case-insensitive file systems How Bazaar operates on case-insensitive file systems such as commonly found on Windows, USB sticks, etc.
- Development repository formats How to work with repository formats that are still under development. Contains instructions for those implementing new formats, of course, but also for (bleeding-edge) end users of those formats.
- Knit pack repositories KnitPack repositories (new in Bazaar 0.92).

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

# **Revision Properties**

Bazaar repositories support setting of a key/value pairs for each revision. Applications can use these properties to store additional information about the revision.

### Usage

In general, revision properties are set by passing keyword argument revprops to method MutableTree.commit. For example:

```
properties = {}
properties['my-property'] = 'test'
tree.commit(message, revprops=properties)
```

Properties can be retrieved via the attribute properties of instances of the class Revision:

```
if 'my-property' in revision.properties:
    my_property = revision.properties['my-property']
    ...
```

# Well-known properties

At the moment, three standardized revision properties are recognized and used by bzrlib:

- authors Authors of the change. This value is a "n" separated set of values in the same format as the committer-id. This property can be set by passing a list to the keyword argument authors of the function MutableTree.commit.
- author Single author of the change. This property is deprecated in favour of authors. It should no longer be set by any code, but will still be read. It is ignored if authors is set in the same revision.
- branch-nick Nickname of the branch. It's either the directory name or manually set by bzr nick. The value is set automatically in MutableTree.commit.
- bugs A list of bug URLs and their statuses. The list is separated by the new-line character (n) and each entry is in format '<URL> <status>'. Currently, bzrlib uses only status 'fixed'. See Bug Trackers for more details about using this feature.

Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

Specifications »

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »



### **Status**

Date: 2007-06-26

bzrlib has a rich API which is used both internally, and externally by plugins and scripts. To allow the API to change, specifically to allow support for features and methods to be removed, without causing hard to diagnose bugs in the clients of the API, bzrlib provides explicit API compatibility data, and a compact API to allow scripts and plugins to ascertain if the bzrlib they are using is compatible to the API they were written against.

#### Contents

- API Versioning
  - Status
  - Motivation
  - Terminology
  - API versions
  - Managing API versions
  - Exported API's
  - Use Cases
    - Requiring bzrlib 0.18 in a plugin
    - Exporting an API from a plugin

# **Motivation**

To allow plugins to apply their own policy for compatibility with bzrlib, without requiring a new release on every library release. Plugins should also be able to use the API to export their own compatibility information for code reuse between plugins.

# Terminology

An **API** is a collection of python objects/modules/packages which can be used by plugins and scripts. The bzrlib **API** covers all of bzrlib, but we can be more precise - e.g. the WorkingTree API. An **API version** is a tuple (major, minor, point).

# **API versions**

For simplicity we treat API's as being compatible with a range of versions: the current release of the API, and some oldest version which is also compatible. While we could say that there is a set of older versions with which the current version is compatible, a range is easier to express, and easier for a human to look at and understand, and finally easier to manage. The oldest version with which the API for a python object is compatible is obtained by looking up the api\_minimum\_version attribute on the python object handed and failing that the bzrlib to require\_api, api minimum version is returned. The current version of the API is obtained by looking for an api\_current\_version attribute, and if that is not found, an version\_info attribute (of which the first 3 elements are used). If no current version can be found, the bzrlib version\_info attribute is used to generate a current API version. This lookup sequence allows users with simple setups (and no python style version\_info tuple) to still export an API version, and for new API's to be managed more granularly later on with a smooth transition everything starts off in lockstep with bzrlib's master version.

API versions are compared lexically to answer the question 'is the requested version  $X \le$  the current version, and  $\ge$  the minimum version'.

# Managing API versions

The minimum API versions should be adjusted to the **oldest** API version with which client code of the API will successfully run. It should not be changed simply because of adding things in a compatible manner, or deprecating features, but rather when errors will occur if client code is not updated. Versions for API's from bzrlib are given the version numbers that bzrlib has had for consistency. Plugins should also take this approach and use the version numbering scheme the plugin used.

# Exported API's

Currently we export a single API - the bzrlib API - and no finer grained APIs. The API versioning support was introduced in bzrlib 0.18. For plugins or tools that want to dynamically check for the presence of the API versioning API, you should compare bzrlib.version\_info[0:3] with (0, 18, 0).

API	Covers	
bzrlib	All of	
DZIIID	bzrlib	
#### **Use Cases**

Some examples of using the API.

```
Requiring bzrlib 0.18 in a plugin
```

```
In the plugins __init__.py:
```

```
import bzrlib
from bzrlib.api import require_api
from bzrlib.errors import IncompatibleAPI
try:
   require_api(bzrlib, (0, 18, 0))
except IncompatibleAPI:
   raise ImportError("A bzrlib compatible with 0.18 is required.
```

#### Exporting an API from a plugin

In the plugin foo exporting the API (in \_\_init\_\_.py):

```
version_info = (0, 0, 1, 'beta', 1)
api_version = (0, 0, 1)
```

In a plugin depending on that plugin (in \_\_init\_\_.py):

```
import bzrlib.plugins.foo
from bzrlib.api import require_api
from bzrlib.errors import IncompatibleAPI
try:
   require_api(bzrlib.plugins.foo, (0, 0, 1))
except IncompatibleAPI:
   raise ImportError("A bzrlib compatible with 0.0.1 is required
```

Specifications »

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

# **Bazaar Apport Integration**

Bazaar can use Apport <<u>http://launchpad.net/apport/</u>> to capture data about unexpected errors (probably, bugs in Bazaar) and report them to the developers.

This is only active for errors that are believed to be internal errors (ie bugs) not user or environmental errors. (See the Developer Guide.)

# Consequences for users

- They shouldn't normally need to see or copy&paste a traceback.
- They will be able to inspect the files before sending them to be sure there's no sensitive data included.
- As at present, they'll need a Launchpad account to report bugs in the normal way.

# Implementation notes

The use of apport by Bazaar is independent of the configuration in the OS. For example in Ubuntu, apport is normally inactive in release builds, and normally excludes software not installed from a package. We'll bypass both of them.

Putting in this handler may mean that an OS-wide exception handler never sees the error, but that was true with our existing exceptionprinter.

The user should have the option to: forget about the crash (and ignore the bug report), see the contents of the report, file a bug, or save the report to file later. At the moment we just show them the filename and let them take it from there.

The process is

- 1. An exception reaches the top-level handler.
- 2. We log it in apport-format to a file in ~/.bazaar/crash.
- 3. We tell the user where that file is, and invite them to file a bug report.

This won't be active for bugs that cause the whole Python interpreter to crash. This can be handled at the OS level. The nice thing is that if apport is active system-wide, it will catch either exceptions in our inprocess apport handler, or errors that crash the intrepreter.

### Future ideas

- Capture apport data even for things not believed to be internal errors, because sometimes they are in fact bugs. Then the user can attach the apport report later if they decide to file a bug. There may be quite a lot of them so we might need to limit the number that are stored, or do this when a debug flag is set. At the moment they go into .bzr.log and that's probably ok to start with.
- Raising an error from the breakin debugger should cause this to fire.
- Developers looking at a crash on their own machine will probably in the first instance just want to see the traceback. Apport files may be more longwinded than our current output and might make the traceback scroll off the screen.
- Automatically trace messages (ie from .bzr.log) in the report. We could just include the whole file, but it may be long, and including the whole thing has a greater risk of including sensitive data.
- Ask the user what they want to do with the report: automatically file it, look at it, see just the traceback, just be told where it is. This could be done through the UIFactory so that it can be done through a graphical dialog.

However, if we've already had an unhandled error in this process there may be problems in Bazaar that prevent us presenting a clean message...

Possibly these bugs are better reported in the next time bzr runs.

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next

Specifications »

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

# Authentication ring

When accessing a remote branch (specified as an URL), it may occur that the server requests an authentication.

This authentication can be provided in different ways:

1. Embedding the user and password in the URL:

bzr branch <scheme>://<user>:<password>@host:port/path

- scheme: Any transport protocol requiring authentication.
- user: The login used to authenticate.
- password: The associated password.
- host: The address of the server.
- port: The port the server is listening to.
- path: The path on the server.

2. Embedding the user in the URL and let bzr find the right password or prompt for one:

bzr branch <scheme>://<user>@host/path

3. Embedding nothing in the URL and let bzr find user and password or prompt for user and/or password:

bzr branch <scheme>://host/path

This specification proposes a mechanism that will allow users to just USE bzr branch <scheme>://host/path Or bzr branch <scheme>://<user>@host/path and leaves bzr find the user and password in its configuration files.

When no user is specified for FTP, SFTP or SSH, the actual behavior of

bzr is to default to getpass.get\_user().

Any implementation of this specification should respect that behaviour.

This specification also proposes a way to describe credentials so that several remote branches can use the same definition. This is particularly important for users handling a lot of passwords and who need to update them on a regular basis.

### Rationale

Embedding user and passwords in the command line is a security hazard (see bug #34685).

Storingpasswordsin~/.bazaar/bazaar.confOr~/.bazaar/locations.confis also a security risk.

Typing user and passwords is error-prone and boring.

Yet, a safe way to store passwords, while allowing bzr to retrieve them, when needed, could improve the bzr user experience.

This specification describes a way to provide user and passwords to bzr while storing them in a relatively safe way.

Note that ssh servers can be configured to use keys instead of (user, password) and, when used with appropriate agents, provide the same kind of comfort this specification aims to provide for all other schemes. Since ssh agents provide a safer way to secure the passwords, this specification is restricted to providing user but does not provide password when used for ssh.

# Authentication definitions

There are two kinds of authentication used by the various schemes supported by bzr:

1. user and password

FTP and SFTP needs a (user, password) to authenticate against a host (SFTP can use ssh keys too, but we don't talk about that in this specification as ssh agents provide a better solution).

2. user, realm and password

HTTP and HTTPS needs a (user, realm, password) to authenticate against a host. But, by using .htaccess files, for example, it is possible to define several (user, realm, password) for a given host. So what is really needed is (user, password, host, path). The realm can be ignored [1] as long as it is still presented to the user when prompting for the password (unless someone found a way to declare two different realms for the same path).

HTTP proxy can be handled as HTTP (or HTTPS) by explicitly specifying the appropriate port.

The true purpose of realms is to allow the same credentials to be reused for disjoint hierarchies. Ignoring them in this specification aims to simplify the user experience while still allowing to share the same credentials for a whole hierarchy.

To take all schemes into account, the password will be deduced from a set of authentication definitions (scheme, host, port, path, user, password).

• scheme: can be empty (meaning the rest of the definition

can be used for any scheme), SFTP and bzr+ssh should not be used here, ssh should be used instead since this is the real scheme regarding authentication,

- host: can be empty (to act as a default for any host),
- port can be empty (useful when an host provides several servers for the same scheme), only numerical values are allowed, this should be used only when the server uses a port different than the scheme standard port,
- path: can be empty (FTP or SFTP will never use it),
- user: can be empty (bzr will defaults to python's getpass.get\_user() for FTP, SFTP and ssh),
- password: can be empty (for security reasons, a user may use the definitions without storing the passwords but want to be prompted ; or the password will be provided by an external plugin via the password\_encoding mechanism decribed below). Must be left empty for ssh.
- password\_encoding: can be empty (default is plaintext).

Also note that an optional verify\_certificates=no field will allow the connection to HTTPS hosts that provides a self certified certificate (the default should be to refuse the connection and inform the user). (Not implemented yet)

Multiple definitions can be provided and, for a given URL, bzr will select a (user [, password]) based on the following rules :

- 1. the first match wins,
- 2. empty fields match everything,
- 3. scheme matches even if decorators are used in the requested URL,
- 4. host matches exactly or act as a domain if it starts with '.' (project.bzr.sf.net will match .bzr.sf.net but projectbzr.sf.net will not match bzr.sf.net).

- 5. port matches if included in the requested URL (exact matches only)
- path matches if included in the requested URL (and by rule #2 above, empty paths will match any provided path).

An optional password\_encoding field may specify how the password is encoded but has no impact on the definition selection.

Possible values are plaintext (no encoding at all) and base64. When the field is absent, plaintext is assumed. Additional encodings may be added in future versions.

Encoding passwords in base64, while weak, provides protection against accidental reading (if an administrator have to look into the file, he will not see the passwords in clear).(Not implemented yet).

This specification intends to ease the authentication providing, not to secure it in the best possible way.

Plugins can provide additional password encodings. The provided netrc\_credential\_store plugin can be used as an example implementation.

Future versions of this specification may provide additional encodings [2].

Additional password encoding methods may be defined that will rely on external means to store the password which, in these cases, will not appear anymore in the definition. It is assumed

[2] that additional password encodings will provide a storage outside of the file described here. The netrc encoding, for example, provides passwords by retrieving them from the .netrc file.

## File format

Even if ~/.bazaar/bazaar.conf and ~/.bazaar/locations.conf seems to provide most of the needed infrastructure, we choose to use a dedicated file for the authentication info ~/.bazaar/authentication.conf for the following reasons:

- allow the user to protect the content of one file only, relaxing security constraints on the others,
- while locations.conf is organized around *local* branches, authentication.conf is organized around *remote* branches or more generally servers. The same authentification definition can even be used for several schemes for servers providing those schemes.

~/.bazaar/authentication.conf will use the same file format as ~/.bazaar/bazaar.conf.

Each section describes an authentication definition.

The section name is an arbitrary string, only the **DEFAULT** value is reserved and should appear as the *last* section.

Each section should define:

• user: the login to be used,

Each section could define:

- host: the remote server,
- port: the port the server is listening,
- verify\_certificates: to control certificate verification (useful for self certified hosts). This applies to HTTPS only.

Accepted values are yes and no, default to yes.

- path: the branch location,
- password: the password,
- password\_encoding: the method used to encode the password if any,

The default content of the file will be:

[DEFAULT]

This section could define:

- user: default user to be used (if not defined the usual bzr way applies, see below).
- password\_encoding: default password encoding.

#### **Use Cases**

The use cases described below use the file format defined above.

 all FTP connections to the foo.net domain are done with the same (user, password):

```
# Identity on foo.net
[foo.net]
scheme=ftp
host=foo.net
user=joe
password=secret-pass
```

will provide ('joe', 'secret-pass') for:

```
bzr branch ftp://foo.net/bzr/branch
bzr pull ftp://bzr.foo.net/bzr/product/branch/trunk
```

• all connections are done with the same user (the remote one for which the default bzr one is not appropriate) and the password is always prompted with some exceptions:

```
# Pet projects on hobby.net
[hobby]
host=r.hobby.net
user=jim
password=obvious1234
# Home server
[home]
scheme=https
host=home.net
user=joe
password='c2VjcmV0LXBhc3M='
password_encoding=base64
verify_certificates=no # Still searching a free certifi
[DEFAULT]
# Our local user is barbaz, on all remote sites we're k
```

user=foobar

• an HTTP server and a proxy:

```
# development branches on dev server
[dev]
scheme=https
host=dev.company.com
path=/dev
user=user1
password=pass1
# toy branches
[localhost]
scheme=http
host=dev.company.com
path=/
user=user2
password=pass2
# proxy
[proxy]
scheme=http
host=proxy.company.com
port=3128
user=proxyuser1
password=proxypass1
```

F

source hosting provider declaring sub-domains for each project:

```
[sfnet domain]
# we use sftp, but ssh is the scheme used for authentic
scheme=ssh
# The leading '.' ensures that 'sf.net' alone doesn't m
host=.sf.net
user=georges
password=ben...son
4]
```

# **UI** Changes

Depending on the info provided in the URL, bzr will interact with the user in different ways:

1. user and password given in the URL.

Nothing to do.

2. user given in the URL.

Get a password from ~/.bazaar/authentication.conf or prompt for one if none is found.

3. No user given in the URL (and no password).

Get a user from ~/.bazaar/authentication.conf or prompt for one if none is found. Continue as 2. (Not implemented yet)

Note: A user will be queried only if the server requires it for HTTP or HTTPS, other protocols always require a user.

In any case, if the server refuses the authentication, bzr reports to the user and terminates.

# Implementation constraints

- bzr should be able to prompt for a user for a given (scheme, host
  [, realm]). Note that realm is available only after a first
  connection attempt to the server.
- No assumptions should be made about the clients of this service (i.e. Transport is the primary target but plugins must be able to use it as well, the definitions used: (scheme, host, [port,] path) are general enough to described credentials for svn servers or LaunchPad xmlrpc calls).
- Policies regarding default users may be taken into account by the implementations, there is no good way to represent that in this specification and stays flexible enough to accommodate various needs (default user policies may differ for different schemes and that may be easier to handle in the code than in the authentication file itself).
- If no user can be found by the mechanism described above, bzr should still default to getpass.get\_user() and may attempt a second matching to obtain a password.
- As this specification proposes a matching between some credentials definitions and real urls, the implementation provides an optional UI feedback about which credential definition is used. Using -Dauth will output some traces in the .bzr.log file metionning the sections used. This allows the user to validate his definitions.

### **Questions and Answers**

- What if a .authinfo file exists ?
  - It will be ignored,
  - Automatic (one-time) conversions may be proposed if sufficient demand exists,
- What if a .netrc file exists ?
  - It is honored if the definition specifies password\_encoding=netrc.
- What mode should the authentication file use ?
  - 600 read/write for owner only by default, if another mode (more permissive) is used, a warning will be issued to inform the users of the potential risks.(Not implemented yet)
- What about using seahorse on Ubuntu or KeyChain Access on Mac OS X ?
  - plugins can be written and registered to handle the associated password\_encoding.
- Could it be possible to encode the whole authentication file with a ssh key ?
  - yes and if the user configure a ssh-agent it will not be queried for pass-phrase every time we want to query the file for a password. But that seems a bit extreme for a first version.(Not implemented yet and may be never)
- Why can't bzr update the authentication file when it queried the user for a password ?
  - a future version may address that but:

1. The user may want to decide which passwords are stored in the file and which aren't.

2. The user should decide if the passwords are encoded (and how) or not (but we may default to base64).

3. The right definition may be hard to get right, but reducing it to (scheme, host, [port,] user, password) may be a good start. I.e. no path so that all paths on the host will match. The user will have to modify it for more complex configurations anyway.

Home   Documentation	Developer Document Catalog (2.2b1) »	previous   next
Specifications »		

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

# **Bundles**

#### Status

Date: 2007-06-19

This document describes the current and future design of the bzr bundle facility.

#### Contents

- Bundles
  - Status
  - Motivation
  - Desired features
  - Historical Design
  - June 2007 Design
  - Future Plans
    - Physical encoding
    - Code reuse

# **Motivation**

Bundles are intended to be a compact binary representation of the changes done within a branch for transmission between users. Bundles should be able to be used easily and seamlessly - we want to avoid having a parallel set of commands to get data from within a bundle.

A related concept is **merge directives** which are used to transmit bzr merge and merge-like operations from one user to another in such a way that the recipient can be sure they get the correct data the initiator desired.

# **Desired features**

- A bundle should be able to substitute for the entire branch in any bzr command that operates on branches in a read only fashion.
- Bundles should be as small as possible without losing data to keep them feasible for including in emails.

## **Historical Design**

Not formally documented, the current released implementation can be found in bzrlib.bundle.serializer. One key element is that this design included parts of the branch data as human readable diffs; which were then subject to corruption by transports such as email.

# June 2007 Design

Bundle Format 4 spec

### Future Plans

Bundles will be implemented as a 'Shallow Branch' with the branch and repository data combined into a single file. This removes the need to special case bundle handling for all command which read from branches.

#### Physical encoding

Bundles will be encoded using the bzr pack format. Within the pack the branch metadata will be serialised as a BzrMetaDir1 branch entry. The Repository data added by the revisions contained in the bundle will be encoded using multi parent diffs as they are the most pithy diffs we are able to create today in the presence of merges. XXX More details needed?

#### Code reuse

Ideally we can reuse our BzrMetaDir based branch formats directly within a Bundle by layering a Transport interface on top of the pack or just copying the data out into a readonly memory transport when we read the pack. This suggests we will have a pack specific Control instance, replacing the usual 'BzrDir' instance, but use the Branch class as-is.

For the Repository access, we will create a composite Repository using the planned Repository Stacking API, and a minimal Repository implementation that can work with the multi parent diffs within the bundle.

We will need access to a branch that has the basis revision of the bundle to be able to construct revisions from within it - this is a requirement for Shallow Branches too, so hopefully we can define a single mechanism at the Branch level to gain access to that.

$\diamondsuit$ Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Specifications »	

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

**Container format** 

#### Status

Date: 2007-06-07

This document describes the proposed container format for streaming and storing collections of data in Bazaar. Initially this will be used for streaming revision data for incremental push/pull in the smart server for 0.18, but the intention is that this will be the basis for much more than just that use case.

In particular, this document currently focuses almost exclusively on the streaming case, and not the on-disk storage case. It also does not discuss the APIs used to manipulate containers and their records.

Contents	
<ul> <li>Container format         <ul> <li>Status</li> </ul> </li> </ul>	
<ul> <li>Motivation</li> <li>Terminology</li> <li>Use Cases</li> </ul>	
<ul> <li>Streaming data between a smart server and clier</li> <li>Incremental push or pull</li> <li>Persistent storage on disk</li> <li>Usable before deep model changes to Bazaar</li> <li>Examples of possible record content</li> <li>Characteristics         <ul> <li>No length-prefixing of entire container</li> <li>Structured as a self-contained series of records</li> <li>Addressing records</li> <li>Reasonably cheap for small records</li> </ul> </li> </ul>	nt

- Record types
  - End Marker
  - Bytes
- Names
## **Motivation**

To create a low-level file format which is suitable for solving the smart server latency problem and whose layout and requirements are extendable in future versions of Bazaar, and with no requirements that the smart server does not have today.

## Terminology

A **container** is a streamable file that contains a series of **records**. Records may have **names**, and consist of bytes.

## Use Cases

Here's a brief description of use cases this format is intended to support.

### Streaming data between a smart server and client

It would be nice if we could combine multiple containers into a single stream by something no more expensive than concatenation (e.g. by omitting end/start marker pairs).

This doesn't imply that such a combination necessarily produces a valid container (e.g. care must be taken to ensure that names are still unique in the combined container), or even a useful container. It is simply that the cost of assembling a new combined container is practically as cheap as simple concatenation.

#### Incremental push or pull

Consider the use case of incremental push/pull, which is currently (0.16) very slow on high-latency links due to the large number of round trips. What we'd like is something like the following.

A client will make a request meaning "give me the knit contents for these revision IDs" (how the client determines which revision IDs it needs is unimportant here). In response, the server streams a single container of:

- one record per file-id:revision-id knit gzip contents and graph data,
- one record per inventory:revision-id knit gzip contents and graph data,
- one record per revision knit gzip contents,
- one record per revision signature,

• end marker record.

in that order.

### Persistent storage on disk

We want a storage format that allows lock-free writes, which suggests a format that uses *rename into place*, and *do not modify after writing*.

### Usable before deep model changes to Bazaar

We want a format we can use and refine sooner rather than later. So it should be usable before the anticipated model changes for Bazaar "1.0" land, while not conflicting with those changes either.

Specifically, we'd like to have this format in Bazaar 0.18.

### Examples of possible record content

- full texts of file versions
- deltas of full texts
- revisions
- inventories
- inventory as tree items e.g. the inventory data for 20 files
- revision signatures
- per-file graph data
- annotation cache

## Characteristics

Some key aspects of the described format are discussed in this section.

### No length-prefixing of entire container

The overall container is not length-prefixed. Instead there is an end marker so that readers can determine when they have read the entire container. This also does not conflict with the goal of allowing single-pass writing.

### Structured as a self-contained series of records

The container contains a series of *records*. Each record is selfdelimiting. Record markers are lightweight. The overhead in terms of bytes and processing for records in this container vs. the raw contents of those records is minimal.

### Addressing records

There is a requirement that each object can be given an arbitrary name. Some version control systems address all content by the SHA-1 digest of that content, but this scheme is unsatisfactory for Bazaar's revision objects. We can still allow addressing by SHA-1 digest for those content types where it makes sense.

Some proposed object names:

- to name a revision: "revision:revision-id". e.g., revision:pqm@pqm.ubuntu.com-20070531210833-8ptk86ocu822hjd5.
- to name an inventory delta: "inventory.delta:revision-id".

e.g., inventory.delta:pqm@pqm.ubuntu.com-20070531210833-8ptk86ocu822hjd5.

It seems likely that we may want to have multiple names for an object. This format allows that (by allowing multiple name headers in a Bytes record).

Although records are in principle addressable by name, this specification alone doesn't provide for efficient access to a particular record given its name. It is intended that separate indexes will be maintained to provide this.

It is acceptable to have records with no explicit name, if the expected use of them does not require them. For example:

- a record's content could be self-describing in the context of a particular container, or
- a record could be accessed via an index based on SHA-1, or
- when streaming, the first record could be treated specially.

### Reasonably cheap for small records

The overhead for storing fairly short records (tens of bytes, rather than thousands or millions) is minimal. The minimum overhead is 3 bytes plus the length of the decimal representation of the *length* value (for a record with no name).

## Specification

This describes just a basic layer for storing a simple series of "records". This layer has no intrinsic understanding of the contents of those records.

The format is:

- a container lead-in, "Bazaar pack format 1 (introduced in 0.18)\n",
- followed by one or more **records**.

A record is:

- a 1 byte kind marker.
- 0 or more bytes of record content, depending on the record type.

### Record types

#### **End Marker**

An End Marker record:

- has a kind marker of "E",
- no content bytes.

End Marker records signal the end of a container.

#### **Bytes**

#### A Bytes record:

• has a kind marker of "B",

 followed by a mandatory content length [1]: "number\n", where number is in decimal, e.g:

1234

• followed by zero or more optional **names**: "name \n", e.g.:

revision:pqm@pqm.ubuntu.com-20070531210833-8ptk86ocu822

- followed by an end of headers byte: "\n",
- followed by some **bytes**, exactly as many as specified by the length prefix header.

So a Bytes record is a series of lines encoding the length and names (if any) followed by a body.

For example, this is a possible Bytes record (including the kind marker):

```
B26
example-name1
example-name2
abcdefghijklmnopgrstuvwxyz
```

#### Names

Names should be UTF-8 encoded strings, with no whitespace. Names should be unique within a single container, but no guarantee of uniqueness outside of the container is made by this layer. Names need to be at least one character long.

This requires that the writer of a record knows the full length of the record up front, which typically means it will need to buffer [1] an entire record in memory. For the first version of this format

this is considered to be acceptable.

$\clubsuit$ Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Specifications »	

#### Home | Documentation | Developer Document Catalog (2.2b1) » previous | next Specifications »

This document contains notes about the design for groupcompress, replacement VersionedFiles store for use in pack based repositories. The goal is to provide fast, history bounded text extraction.

## Overview

The goal: Much tighter compression, maintained automatically. Considerations to weigh: The minimum IO to reconstruct a text with no other repository involved; The number of index lookups to plan a reconstruction. The minimum IO to reconstruct a text with another repositories assistance (affects network IO for fetch, which impacts incremental pulls and shallow branch operations).

## Current approach

Each delta is individually compressed against another text, and then entropy compressed. We index the pointers between these deltas.

Solo reconstruction: Plan a readv via the index, read the deltas in forward IO, apply each delta. Total IO: sum(deltas) + deltacount\*index overhead. Fetch/stacked reconstruction: Plan a readv via the index, using local basis texts where possible. Then readv locally and remote and apply deltas. Total IO as for solo reconstruction.

### Things to keep

Reasonable sizes 'amount read' from remote machines to reconstruct an arbitrary text: Reading 5MB for a 100K plain text is not a good trade off. Reading (say) 500K is probably acceptable. Reading ~100K is ideal. However, its likely that some texts (e.g NEWS versions) can be stored for nearly-no space at all if we are willing to have unbounded IO. Profiling to set a good heuristic will be important. Also allowing users to choose to optimise for a server environment may make sense: paying more local IO for less compact storage may be useful.

## Things to remove

Index scatter gather IO. Doing hundreds or thousands of index lookups is very expensive, and doing that per file just adds insult to injury.

Partioned compression amongst files.

Scatter gather IO when reconstructing texts: linear forward IO is better.

### Thoughts

Merges combine texts from multiple versions to create a new version. Deltas add new text to existing files and remove some text from the same. Getting high compression means reading some base and then a chain of deltas (could be a tree) to gain access to the thing that the final delta was made against, and that delta. Rather than composing all these deltas, we can just just perform the final diff against the base text and the serialised invidual deltas. If the diff algorithm can reuse out of order lines from previous texts (e.g. storing AB -> BA as pointers rather than delete and add, then the presence of any previously stored line in a single chain can be reused. One such diff algorithm is xdelta, another reasonable one to consider is plain old zlib or Izma. We could also use bzip2. One advantage of using a generic compression engine is less python code. One advantage of preprocessing line based deltas is that we reduce the window size for the text repeated within lines, and that will help compression by a simple entropy compressor as a post processor. Izma appears fantastic at compression - 420MB of NEWS files down to 200KB. so window size appears to be a key determiner for efficiency.

# Delta strategy

Very big objects - no delta. I plan to kick this in at 5MB initially, but once the codebase is up and running, we can tweak this to

Very small objects - no delta? If they are combined with a larger zlib object why not? (Answer: because zlib's window is really small)

Other objects - group by fileid (gives related texts a chance, though using a file name would be better long term as e.g. COPYING and COPYING from different projects could combine). Then by reverse topological graph(as this places more recent texts at the front of a chain). Alternatively, group by size, though that should not matter with a large enough window. Finally, delta the texts against the current output of the compressor. This is essentially a somewhat typed form of sliding window dictionary compression. An alternative implementation would be to just use zlib, or lzma, or bzip2 directory.

Unfortunately, just using entropy compression forces a lot of data to be output by the decompressor - e.g. 420MB in the NEWS sample corpus. When we only want a single 55K text thats inefficient. (An initial test took several seconds with Izma.)

The fastest to implement approach is probably just 'diff output to date and add to entropy compressor'. This should produce reasonable results. As delta chain length is not a concern (only one delta to apply ever), we can simply cap the chain when the total read size becomes unreasonable. Given older texts are smaller we probably want some weighted factor of plaintext size.

In this approach, a single entropy compressed region is read as a unit, giving the lower bound for IO (and how much to read is an open question - what byte offset of compressed data is sufficient to ensue that the delta-stream contents we need are reconstructable. Flushing, while possible, degrades compression(and adds overhead - we'd be paying 4 bytes per record guaranteed). Again - tests will be needed.

A nice possibility is to output mpdiff compatible records, which might enable some code reuse. This is more work than just diff (current\_out, new\_text), so can wait for the concept to be proven.

# Implementation Strategy

Bring up a VersionedFiles object that implements this, then stuff it into a repository format. zlib as a starting compressor, though bzip2 will probably do a good job.

Home   Documentation	Developer Document Catalog (2.2b1) »	previous   next
Specifications »		

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

# Indices

### **Status**

#### Date: 2007-07-14

This document describes the indexing facilities within bzrlib.

# Contents

- Indices
  - Status
  - Motivation
  - Terminology
  - Overview
  - General Index API
    - Services
      - Build index
      - Retrieve entries from the index
      - Merging of indices
  - Index implementations
    - GraphIndex

## **Motivation**

To provide a clean concept of index that can be reused by different components within the codebase rather than being rewritten every time by different components.

## Terminology

An **index** is a dictionary mapping opaque keys to opaque values. Different index types may allow some of the value data to be interpreted by the index. For example the GraphIndex index stores a graph between keys as part of the index.

## **Overview**

bzr is moving to a write-once model for repository storage in order to achieve lock-free repositories eventually. In order to support this, we are making our new index classes **immutable**. That is, one creates a new index in a single operation, and after that it is read only. To combine two indices a Combined\* index may be used, or an **index merge** may be performed by reading the entire value of two (or more) indices and writing them into a new index.

### **General Index API**

We may end up with multiple different Index types (e.g. GraphIndex, Index, WhackyIndex). Even though these may require different method signatures to operate would strive to keep the signatures and return values as similar as possible. e.g.:

```
GraphIndexBuilder - add_node(key, value, references)
IndexBuilder - add_node(key, value)
WhackyIndexBuilder - add_node(key, value, whackiness)
```

as opposed to something quite different like:

```
node = IncrementalBuilder.get_node()
node.key = 'foo'
node.value = 'bar'
```

### Services

An initial implementation of indexing can probably get away with a small number of primitives. Assuming we have write once index files:

#### **Build index**

This should be done by creating an IndexBuilder and then calling insert(key, value) many times. (Indices that support sorting, topological sorting etc, will want specialised insert methods).

When the keys have all been added, a finish method should be called, which will return a file stream to read the index data from.

#### Retrieve entries from the index

This should allow random access to the index using ready, so we

probably want to open the index on a Transport, then use iter\_entries(keys), which can return an iterator that yields (key, value) pairs in whatever order makes sense for the index.

#### Merging of indices

Merging of N indices requires a concordance of the keys of the index. So we should offer a iter\_all\_entries call that has the same return type as the iter\_entries call.

## Index implementations

### GraphIndex

GraphIndex supports graph based lookups. While currently unoptimised for reading, the index is quite space efficient at storing the revision graph index for bzr. The GraphIndexBuilder may be used to create one of these indices by calling add\_node until all nodes are added, then finish to obtain a file stream containing the index data. Multiple indices may be queried using the CombinedGraphIndex Class.

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next Specifications »

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

## **Inventories**

#### Contents

- Inventories
  - Overview
  - In memory inventories
  - Serialization
    - dirstate
    - xml
  - Serialization scaling and future designs
    - Current situation
    - Long term work
    - Layering
    - Design elements to achieve the goals in a future inventory implementation
  - Hash bucket based inventories
    - Overview
    - Goal satisfaction
    - Issues
    - Canonical form
    - Apply
    - Delta
  - Radix tree based inventories
    - Overview
    - Goal satisfaction
    - Issues
    - Canonical form
    - Apply
    - Delta
  - Hash Trie details
    - Insertion
  - Inventory deltas

• Delta consistency

Avoiding inconsistent deltas

## Overview

Inventories provide an abstraction for talking about the shape of a tree. Generally only tree object implementors should be concerned about entire inventory objects and their implementation. Other common exceptions are full-tree operations such as 'checkout', 'export' and 'import'.

## In memory inventories

In memory inventories are often used in diff and status operations between trees. We are working to reduce the number of times this occurs with 'full tree' inventory objects, and instead use more custom tailored data structures that allow operations on only a small amount of data regardless of the size of the tree.

## Serialization

There are several variants of serialised tree shape in use by bzr. To date these have been mostly xml based, though plugins have offered non-xml versions.

### dirstate

The dirstate file in a working tree includes many different tree shapes - one for the working tree and one for each parent tree, interleaved to allow efficient diff and status operations.

### xml

All the xml serialized forms write to and read from a single byte string, whose hash is then the inventory validator for the commit object.

## Serialization scaling and future designs

Overall efficiency and scaling is constrained by the bottom level structure that an inventory is stored as. We have a number of goals we want to achieve:

- 1. Allow commit to write less than the full tree's data in to the repository in the general case.
- 2. Allow the data that is written to be calculated without examining every versioned path in the tree.
- 3. Generate the exact same representation for a given inventory regardless of the amount of history available.
- 4. Allow in memory deltas to be generated directly from the serialised form without upcasting to a full in-memory representation or examining every path in the tree. Ideally the work performed will be proportional to the amount of changes between the trees being compared.
- 5. Allow fetch to determine the file texts that need to be pulled to ensure that the entire tree can be reconstructed without having to probe every path in the tree.
- 6. Allow bzr to map paths to file ids without reading the entire serialised form. This is something that is used by commands such as merge PATH and diff -r X PATH.
- 7. Let bzr map file ids to paths without reading the entire serialised form. This is used by commands that are presenting output to the user such as loggerhead, bzr-search, log FILENAME.
- 8. We want a strong validator for inventories which is cheap to generate. Specifically we should be able to create the generator for a new commit without processing all the data of the basis commit.
- 9. Testaments generation is currently size(tree), we would like to create a new testament standard which requires less

work so that signed commits are not significantly slower than regular commits.

We have current performance and memory bugs in log -v, merge, commit, diff -r, loggerhead and status -r which can be addressed by an inventory system meeting these goals.

### Current situation

The xml based implementation we use today layers the inventory as a bytestring which is stored under a single key; the bytestring is then compressed as a delta against the bytestring of its left hand parent by the knit code.

Gap analysis:

- 1. Succeeds
- 2. Fails generating a new xml representation needs full tree data.
- 3. Succeeds the inventory layer accesses the bytestring, which is deterministic
- 4. Fails we have to reconstruct both inventories as trees and then delta the resulting in memory objects.
- 5. Partial success the revision field in the inventory can be scanned for in both text-delta and full-bytestring form; other revision values than those revisions which are being pulled are by definition absent.
- 6. Partially succeeds with appropriate logic a path<->id map can be generated just-in-time, but it is complex and still requires reconstructing the entire byte-string.
- 7. As for 6.
- 8. Fails we have to hash the entire tree in serialised form to generate validators.
- 9. Fails.

### Long term work

Some things are likely harder to fix incrementally than others. In particular, goal 3 (constant canonical form) is arguably only achieved if we remove all derived data such as the last-modified revision from the inventory itself. That said, the last-modified appears to be in a higher level than raw serialization. So in the medium term we will not alter the contents of inventories, only the way that the current contents are mapped to and from disk.

#### Layering

We desire clear and clean layers. Each layer should be as simple as we can make it to aid in debugging and performance tuning. So where we can choose to either write a complex layer and something simple on top of it, or two layers with neither being as complex - then we should consider the latter choice better in the absence of compelling reasons not to.

Some key layers we have today and can look at using or tweaking are:

- Tree objects the abstract interface bzrlib code works in
- VersionedFiles the optionally delta compressing key->bytes storage interface.
- Inventory the abstract interface that many tree operations are written in.

These layers are probably sufficient with minor tweaking. We may want to add additional modules/implementations of one or more layers, but that doesn't really require new layers to be exposed.

# Design elements to achieve the goals in a future inventory implementation
- Split up the logical document into smaller serialised fragements. For instance hash buckets or nodes in a tree of some sort. By serialising in smaller units, we can increase the number of smaller units rather than their size as the tree grows; as long as two similar trees have similar serialised forms, the amount of different content should be quite high.
- Use fragment identifiers that are independent of revision id, so that serialisation of two related trees generates overlap in the keyspace for fragments without requiring explicit delta logic. Content Hash Keys (e.g. ('sha1:ABCDEF0123456789...',) are useful here because of the ability to assign them without reference to history.)
- Store the fragments in our existing VersionedFiles store. Adding an index for them. Have the serialised form be uncompressed utf8, so that delta logic in the VersionedFiles layer can be used. We may need to provide some sort of hinting mechanism to get good compression - but the trivially available zlib compression of knits-with-no-deltas is probably a good start.
- Item\_keys\_introduced\_by is innately a history-using function; we can reproduce the text-key finding logic by doing a tree diff between any tree and an older tree that will limit the amount of data we need to process to something proportional to the difference and the size of each fragment. When checking many versions we can track which fragments we have examined and only look at new unique ones as each version is examined in turn.
- Working tree to arbitrary history revision deltas/comparisons can be scaled up by doing a two-step (fixed at two!) delta combining delta(tree, basis) and then combine that with delta(basis, arbitrary\_revision) using the repositories ability to get a delta cheaply.
- The key primitives we need seem to be: \* canonical\_form(inventory) -> fragments \* delta(inventory,

inventory) -> inventory\_delta \* apply(inventory\_delta, canonical\_form) -> fragments

- Having very many small fragments is likely to cause a high latency multiplier unless we are careful.
- Possible designs to investigate a hash bucket approach, radix trees, B+ trees, directory trees (with splits inside a directory?).

# Hash bucket based inventories

## Overview

We store two maps - fileid:inventory\_entry and path:fileid, in a stable hash trie, stored in densly packed fragments. We pack keys into the map densely up the tree, with a single canonical form for any given tree. This is more stable than simple fixed size buckets, which prevents corner cases where the tree size varies right on a bucket size border. (Note that such cases are not a fatal flaw - the two forms would both be present in the repository, so only a small amount of data would be written at each transition - but a full tree reprocess would be needed at each tree operation across the boundary, and thats undesirable.)

## Goal satisfaction

- 1. Success
- 2. Success
- 3. Success
- 4. Success, though each change will need its parents looked up as well so it will be proportional to the changes + the directories above the changed path.
- 5. Success looking at the difference against all parents we can determine new keys without reference to the repository content will be inserted into.
- 6. This probably needs a path->id map, allowing a 2-step lookup.
- 7. If we allocate buckets by hashing the id, then this is succeed, though, as per 4 it will need recursive lookups.
- 8. Success
- 9. Fail data beyond that currently included in testaments is

included in the strong validator.

#### Issues

1. Tuning the fragment size needs doing. 1. Testing. 1. Writing code. 1. Separate root node, or inline into revision? 1. Cannot do 'Is' efficiently in the current design. 1. Cannot detect invalid deltas easily. 1. What about LCA merge of inventories?

#### Canonical form

There are three fragment types for the canonical form. Each fragment is addressed using a Content Hash Key (CHK) - for instance "sha1:12345678901234567890".

root\_node: (Perhaps this should be inlined into the revision object). HASH\_INVENTORY\_SIGNATURE path\_map: CHK to root of path to id map content\_map: CHK to root of id to entry map

map\_node: INTERNAL\_NODE or LEAF\_NODE INTERNAL\_NODE: INTERNAL\_NODE\_SIGNATURE hash\_prefix: PREFIX prefix\_width: INT PREFIX CHK TYPE SIZE PREFIX CHK TYPE SIZE ...

(Where TYPE is I for internal or L for leaf).

leaf\_node: LEAF\_NODE\_SIGNATURE hash\_prefix: PREFIX HASHx00KEYx00 VALUE

For path maps, VALUE is::

fileid

For content maps, VALUE::

fileid basename kind last-changed kind-specific-details

The path and content maps are populated simply by serialising every inventory entry and inserting them into both the path map and the content map. The maps start with just a single leaf node with an empty prefix.

## Apply

Given an inventory delta - a list of (old\_path, new\_path, InventoryEntry) items, with a None in new\_path indicating a delete operation, and recursive deletes not being permitted - all entries to be deleted must be explicitly listed, we can transform a current inventory directly. We can't trivially detect an invalid delta though.

To perform an application, naively we can just update both maps. For the path map we would remove all entries where the paths in the delta do not match, then insert those with a new\_path again. For the content map we would just remove all the fileids in the delta, then insert those with a new\_path that is not None.

#### Delta

To generate a delta between two inventories, we first generate a list of altered fileids, and then recursively look up their parents to generate their old and new file paths.

To generate the list of altered file ids, we do an entry by entry comparison of the full contents of every leaf node that the two inventories do not have in common. To do this, we start at the root node, and follow every CHK pointer that is only in one tree. We can then bring in all the values from the leaf nodes and do a set difference to get the altered ones, which we would then parse.

# Radix tree based inventories

#### Overview

We store two maps - fileid:path and path:inventory\_entry. The fileid:path map is a hash trie (as file ids have no useful locality of reference). The path:inventory\_entry map is stored as a regular trie. As for hash tries we define a single canonical representation for regular tries similar to that defined above for hash tries.

#### Goal satisfaction

- 1. Success
- 2. Success
- 3. Success
- 4. Success
- 5. Success looking at the difference against all parents we can determine new keys without reference to the repository content will be inserted into.
- 6. Success
- 7. Success
- 8. Success
- 9. Fail data beyond that currently included in testaments is included in the strong validator.

#### Issues

1. Tuning the fragment size needs doing. 1. Testing. 1. Writing code. 1. Separate root node, or inline into revision? 1. What about LCA merge of inventories?

Canonical form

There are five fragment types for the canonical form:

The root node, hash trie internal and leaf nodes as previous.

Then we have two more, the internal and leaf node for the radix tree.

radix\_node: INTERNAL\_NODE or LEAF\_NODE

INTERNAL\_NODE: INTERNAL\_NODE\_SIGNATURE prefix: PREFIX suffix CHK TYPE SIZE suffix CHK TYPE SIZE ...

(Where TYPE is I for internal or L for leaf).

LEAF\_NODE: LEAF\_NODE\_SIGNATURE prefix: PREFIX suffixx00VALUE

For the content map we use the same value as for hashtrie inventories.

Node splitting and joining in the radix tree are managed in the same fashion as as for the internal nodes of the hashtries.

## Apply

Apply is implemented as for hashtries - we just remove and reinsert the fileid:paths map entries, and likewise for the path:entry map. We can however cheaply detect invalid deltas where a delete fails to include its children.

## Delta

Delta generation is very similar to that with hash tries, except we get the path of nodes as part of the lookup process.

## Hash Trie details

The canonical form for a hash trie is a tree of internal nodes leading down to leaf nodes, with no node exceeding some threshold size, and every node containing as much content as it can, but no leaf node containing less than its lower size threshold. (In the event that an imbalance in the hash function causes a tree where an internal node is needed, but any prefix generates a child with less than the lower threshold, the smallest prefix should be taken). An internal node holds some number of key prefixes, all with the same bit-width. A leaf node holds the actual values. As trees do not spring fullyformed, the canonical form is defined iteratively - by taking every item in a tree and inserting it into a new tree in order you can determine what canonical form would look like. As that is an expensive operation, it should only be done rarely.

Updates to a tree that is in canonical form can be done preserving canonical form if we can prove that our rules for insertion are orderindependent, and that our rules for deletion generate the same tree as if we never inserted those nodes.

Our hash tries are balanced vertically but not horizontally. That is, one leg of a tree can be arbitrarily deeper than adjacent legs. We require that each node along a path within the tree be densely packed, with the densest nodes near the top of the tree, and the least dense at the bottom. Except where the tree cannot support it, no node is smaller than a minimum\_size, and none larger than maximum\_size. The minimum size constraint is only applied when there are enough entries under a prefix to meet that minimum. The maximum size constraint is always applied except when a node with a single entry is larger than the maximum size constraint, and if the minimum size contraint is to be ignored, a deeper prefix can be

chosen to pack the containing node more densely, as long as no additional minimum sizes checks on child nodes are violated.

## Insertion

- 1. Hash the entry, and insert the entry in the leaf node with a matching prefix, creating that node and linking it from the internal node containing that prefix if there is no appropriate leaf node.
- Starting at the highest node altered, for all altered nodes, check if it has transitioned across either size boundary - 0 < min\_size < max\_size. If it has not, proceed to update the CHK pointers.
- 3. If it increased above min\_size, check the node above to see if it can be more densely packed. To be below the min\_size the node's parent must have hit the max size constraint and been forced to split even though this child did not have enough content to support a min\_size node so the prefix chosen in the parent may be shorter than desirable and we may now be able to more densely pack the parent by splitting the child nodes more. So if the parent node can support a deeper prefix without hitting max\_size, and the count of under min\_size nodes cannot be reduced, the parent should be given a deeper prefix.
- 4. If it increased above max\_size, shrink the prefix width used to split out new nodes until the node is below max\_size (unless the prefix width is already 1 the minimum). To shrink the prefix of an internal node, create new internal nodes for each new prefix, and populate them with the content of the nodes which were formerly linked. (This will normally bubble down due to keeping densely packed nodes). To shrink the prefix of a leaf node, create an internal node with the same prefix, then choose a width for the internal node such that the contents of the leaf all fit into new leaves obeying the min\_size and max\_size rules. The largest prefix possible should be chosen, to obey the higher-nodes-are-denser rule. That rule also gives room in leaf

nodes for growth without affecting the parent node packing.

5. Update the CHK pointers - serialise every altered node to generate a CHK, and update the CHK placeholder in the nodes parent; then reserialise the parent. CHK pointer propagation can be done lazily when many updates are expected.

Multiple versions of nodes for the same PREFIX and internal prefix width should compress well for the same tree.

## Inventory deltas

An inventory is a serialization of the in-memory inventory delta. To serialize an inventory delta, one takes an existing inventory delta and the revision\_id of the revision it was created it against and the revision id of the inventory which should result by applying the delta to the parent. We then serialize every item in the delta in a simple format:

'format: bzr inventory delta v1 (1.14)' NL 'parent:' SP BASIS\_INVENTORY NL 'version:' SP NULL\_OR\_REVISION NL 'versioned\_root:' SP BOOL NL 'tree\_references:' SP BOOL NL DELTA\_LINES

DELTA LINES ::= (DELTA\_LINE NL)\* DELTA\_LINE ::= OLDPATH NULL NEWPATH file-id NULL PARENT ID NULL NULL LAST MODIFIED NULL CONTENT SP ::= ' ' BOOL ::= 'true' | 'false' NULL ::= x00 OLDPATH ::= NONE | PATH NEWPATH ::= NONE | PATH NONE ::= 'None' PATH ::= path PARENT ID ::= FILE ID | '' DELETED CONTENT FILE CONTENT CONTENT ::= T TREE CONTENT LINK CONTENT DIR CONTENT DELETED CONTENT ::= 'deleted' FILE CONTENT ::= 'file' NULL text size NULL EXEC NULL text sha1 DIR CONTENT ::= 'dir' TREE CONTENT ::= 'tree' NULL tree-revision LINK CONTENT ::= 'link' NULL link-target BASIS INVENTORY ::= NULL OR REVISION LAST MODIFIED ::= NULL OR REVISION NULL OR REVISION ::= 'null:' | REVISION REVISION ::= revisionid-in-utf8-no-whitespace EXEC ::= '' | 'Y'

DELTA\_LINES is lexicographically sorted.

Some explanation is in order. When NEWPATH is 'None' a delete has been recorded, and because this inventory delta is not attempting to be a reversible delta, the only other valid fields are OLDPATH and 'file-id'. PARENT\_ID is '' when a delete has been recorded or when recording a new root entry.

## Delta consistency

Inventory deltas and more broadly changes between trees are a significant part of bzr's core operations: they are key components in status, diff, commit, and merge (although merge uses tree transform, deltas contain the changes that are applied to the transform). Our ability to perform a given operation depends on us creating consistent deltas between trees. Inconsistent deltas lead to errors and bugs, or even just unexpected conflicts.

An inventory delta is a transform to change an inventory A into another inventory B (in patch terms its a perfect patch). Sometimes, for instance in a regular commit, inventory B is known at the time we create the delta. Other times, B is not known because the user is requesting that some parts of the second inventory they have are masked out from consideration. When this happens we create a delta that when applied to A creates a B we haven't seen in total before. In this situation we need to ensure that B will be internally consistent. Deltas are unidirectional, a delta(A, B) creates B from A, but cannot be used to create A from B.

Deltas are expressed as a list of (oldpath, newpath, fileid, entry) tuples. The fileid, entry elements are normative; the old and new paths are strong hints but not currently guaranteed to be accurate. (This is a shame and something we should tighten up). Deltas are required to list all removals explicitly - removing the parent of an entry doesn't remove the entry.

Applying a delta to an inventory consists of:

- removing all fileids for which entry is None
- adding or replacing all other fileids
- detecting consistency errors

An interesting aspect of delta inconsistencies is when we notice

them:

- Silent errors which our application logic misses
- Visible errors we catch during application, so bad data isn't stored in the system.

The minimum safe level for our application logic would be to catch all errors during application. Making generation never generate inconsistent deltas is a seperate but necessary condition for robust code.

An inconsistent delta is one which:

- after application to an inventory the inventory is an impossible state.
- has the same fileid, or oldpath(not-None), or newpath(not-None) multiple times.
- has a fileid field different to the entry.fileid in the same item in the delta.
- has an entry that is in an impossible state (e.g. a directory with a text size)

Forms of inventory inconsistency deltas can carry/cause:

- An entry newly introduced to a path without also removing or relocating any existing entry at that path. (Duplicate paths)
- An entry whose parent id isn't present in the tree. (Missing parent).
- Having oldpath or newpath not be actual original path or resulting path. (Wrong path)
- An entry whose parent is not a directory. (Under non-directory).
- An entry that is internally inconsistent.
- An entry that is already present in the tree (Duplicate id)

Known causes of inconsistency:

• A 'new' entry which the inventory already has - when this is a directory even arbitrary file ids under the 'new' entry are more likely to collide on paths.

- Removing a directory without recursively removing its children causes Missing parent.
- Recording a change to an entry without including all changed entries found following its parents up to and includin the root
   can cause duplicate paths, missing parents, wrong path, under non-directory.

## Avoiding inconsistent deltas

The simplest thing is to never create partial deltas, as it is trivial to be consistent when all data is examined every time. However users sometimes want to specify a subset of the changes in their tree when they do an operation which needs to create a delta - such as commit.

We have a choice about handling user requests that can generate inconsistent deltas. We can alter or interpret the request in such a way that the delta will be consistent, but perhaps larger than the user had intended. Or we can identify problematic situations and abort, specifying to the user why we have aborted and likely things they can do to make their request generate a consistent delta.

Currently we attempt to expand/interpret the request so that the user is not required to understand all the internal constraints of the system: if they request 'foo/bar' we automatically include foo. This works but can surprise the user sometimes when things they didn't explicitly request are committed.

Different trees can use different algorithms to expand the request as long as they produce consistent deltas. As part of getting a consistent UI we require that all trees expand the paths requested downwards. Beyond that as long as the delta is consistent it is up to the tree.

Given two trees, source and target, and a set of selected file ids to

check for changes and if changed in a delta between them, we have to expand that set by the following rules, to get consistent deltas. The test for consistency is that if the resulting delta is applied to source, to create a third tree 'output', and the paths in the delta match the paths in source and output, only one file id is at each path in output, and no file ids are missing parents, then the delta is consistent.

Firstly, the parent ids to the root for all of the file ids that have actually changed must be considered. Unless they are all examined the paths in the delta may be wrong.

Secondly, when an item included in the delta has a new path which is the same as a path in source, the fileid of that path in source must be included. Failing to do this leads to multiple ids tryin to share a path in output.

Thirdly, when an item changes its kind from 'directory' to anything else in the delta, all of the direct children of the directory in source must be included.

$\clubsuit$ Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Specifications »	

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »



by Aaron Bentley

# **Essential characteristics**

In the general case (no criss-cross), it is a three-way merge. When there is a criss-cross at the tree level, but not for the particular file, it is still a three-way merge. When there's a file-level criss-cross, it's superior to a three-way merge.

# Algorithm

First, we compare the files we are trying to merge, and find the lines that differ. Next, we try to determine why they differ; this is essential to the merge operation, because it affects how we resolve the differences. In this merger, there are three possible outcomes:

- 1. The line was added in this version: "new-this"
- 2. The line was deleted in the other version: "killed-other"
- 3. The line was preserved as part of merge resolution in this version, but deleted in the other version: "conflicted-this"

Option 3 is new, but I believe it is essential. When each side has made a conflicting merge resolution, we should let the user decide how to combine the two resolutions, i.e. we should emit a conflict. We cannot silently drop the line, or silently keep the line, which can happen if we choose options 1 or 2. If we choose options 1 or 2, there's also a possibility that a conflict will be produced, but no guarantee. We need a guarantee, which is why we need a new possible outcome.

To decide whether a line is "new-this", "killed-other" or "conflictedthis", we compare this version against the versions from each "least common ancestor" (LCA), in graph terminology. For each LCA version, if the line is not present in the LCA version, we add it to the "new" set. If the line is present in the LCA version, we add it to the "killed" set.

When we are done going through each LCA version, each unique line will be in at least one of the sets. If it is only in the "new" set, it's handled as "new-this". If it is only in the "killed" set, it's handled as "killed-other". If it is in both sets, it's handled as "conflicted-this".

The logic here is a bit tricky: first, we know that the line is present in

some, but not all, LCAs. We can assume that all LCAs were produced by merges of the same sets of revisions. That means that in those LCAs, there were different merge resolutions. Since THIS and OTHER disagree about whether the line is present, those differences have propagated into THIS and OTHER. Therefore, we should declare that the lines are in conflict, and let the user handle the issue.

## LCA merge and Three-way merge

Now, in the common case, there's a single LCA, and LCA merge behaves as a three-way merge. Since there's only one LCA, we cannot get the "conflicted-this" outcome, only "new-this" or "killedother. Let's look at the typical description of three-way merges:

THIS	BASE	OTHER	OUT
A	А	А	A
A	В	А	A
A	В	В	A
A	А	В	В
A	В	С	*conflict*

Now, let's assume that BASE is a common ancestor, as is typically the case. In fact, for best-case merges, BASE is the sole LCA.

We always pick the version that represents a change from BASE, if there is one. For the AAAA line, there is no change, so the output is rightfully BASE/THIS/OTHER. For ABAA, the THIS and OTHER are changes from BASE, and they are the same change so they both win. (This case is sometimes called convergence.) For ABBA, THIS is a change from BASE, so THIS wins. For AABB, OTHER is a change from BASE, so OTHER wins. For ABC\*, THIS and OTHER are both changes to BASE, but they are different changes, so they can't both win cleanly. Instead, we have a conflict.

Now in three-way merging, we typically talk about regions of text. In weave/knit/newness/lca merge, we also have regions. Each contiguous group of "unchanged" lines is a region, and the areas between them are also regions.

Let's assign a to THIS and b to OTHER. "unchanged" regions represent the AAAA or ABAA cases; it doesn't matter which,

because the outcome is the same regardless. Regions which consist of only "new-a" or "killed-a" represent the ABBA case. Regions which consist of only "new-b" or "killed-b" represent the AABB case. Regions which have (new-a or killed-a) AND (new-b or killed-b) are the ABC\* case— both sides have made changes, and they are different changes, so a conflict must be emitted.

This is what I mean when I say that it is a three-way merge in the common case; if there is only one LCA, then it is merely an alternative implementation of three-way. (One that happens to automatically do --reprocess, ftw).

## Exception to three-way behavior

There is a special case of three-way merge which LCA merge handles differently from our default "merge3" algorithm: BASE has content X, THIS deletes the content, and OTHER changes X to Y. In this case, LCA merge emits Y in its output and does not indicate a conflict. merge3 would output Y, but would also indicate a conflict. (This is also the behavior in the inverse case where OTHER has nothing and THIS has Y.)

This behavior is due the way LCA determines basic conflicts; they can only be emitted when THIS and OTHER each have unique lines between common lines. If THIS does not have unique lines in this position, conflicts will not be emitted, even if its (lack of) content is unique.

This behavior difference is shared with "weave" merge. I hope that a future revision of LCA merge will handle this case as merge3 would.

## Why a new name

- It was time. Although knit / annotate merge and newness merge have tried to emulate the behavior of the original weave merge algorithm, --merge-type=weave hasn't been based on weaves for a long time.
- Behavior differences. This algorithm should behave like a threeway merge in the common case, while its predecessors did not. It also has explicit support for handling conflicting merge resolutions, so it should behave better in criss-cross merge scenarios.

## Performance

Unlike the current "weave" merge implementation, Ica merge does not perform any whole-history operations. LCA selection should scale with the number of uncommon revisions. Text comparison time should scale mO(n<sup>2</sup>), where m is the number of LCAs, and n is the number of lines in the file. The current weave merge compares each uncommon ancestor, potentially several times, so it is >=  $kO(n^2)$ , where k is the number of uncommon ancestors. So "Ica" should beat "weave" both in history analysis time and in text comparison time.

# Possible flaws

- Inaccurate LCA selection. Our current LCA algorithm uses Graph.heads(), which is known to be flawed. It may occasionally give bad results. This risk is mitigated by the fact that the per-file graphs tend to be simpler than the revision graph. And since we're already using this LCA algorithm, this is not an additional risk. I hope that John Meinel will soon have a fixed version of Graph.heads for us.
- 2. False matches. Weaves have a concept of line identity, but knits and later formats do not. So a line may appear to be common to two files, when in fact it was introduced separately into each for entirely different reasons. This risk is the same for three-way merging. It is mitigated by using Patience sequence matching, which a longest-common-subsequence match.

# Acknowledgements

I think this could be a great merge algorithm, and a candidate to make our default, but this work would not have been possible without the work of others, especially:

- Martin Pool's weave merge and knit/annotate merge algorithms.
- Bram Cohen's discussions of merge algorithms
- Andrew Tridgell's dissection of BitKeeper merge
- Nathaniel Smith's analysis of why criss-cross histories necessarily produce poor three-way merges.

ig  Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Specifications »	

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

# **Network Protocol**

Date: 2009-01-07

#### Contents

- Network Protocol
  - Overview
  - Layering
    - Medium
    - Protocol
    - Request/Response processing
    - Server-side
    - Client-side
  - Protocol description
    - Version one
    - Version two
    - Version two with streamed bodies
    - Version three
      - Headers
      - Conventional requests and responses
      - Early error returns
      - Full-duplex operation
  - APIs
  - Paths
  - Requests
  - Recognised errors

## Overview

The smart protocol provides a way to send a requests and corresponding responses to communicate with a remote bzr process.

# Layering

## Medium

At the bottom level there is either a socket, pipes, or an HTTP request/response. We call this layer the *medium*. It is responsible for carrying bytes between a client and server. For sockets, we have the idea that you have multiple requests and get a read error because the other side did shutdown. For pipes we have read pipe which will have a zero read which marks end-of-file. For HTTP server environment there is no end-of-stream because each request coming into the server is independent.

So we need a wrapper around pipes and sockets to separate out requests from substrate and this will give us a single model which is consistent for HTTP, sockets and pipes.

#### Protocol

On top of the medium is the *protocol*. This is the layer that deserialises bytes into the structured data that requests and responses consist of.

#### **Request/Response processing**

On top of the protocol is the logic for processing requests (on the server) or responses (on the client).

#### Server-side

Sketch:

MEDIUM (factory for protocol, reads bytes & pushes to protocol

```
uses protocol to detect end-of-request, sends written
         bytes to client) e.g. socket, pipe, HTTP request handl
 Λ
 | bytes.
 v
PROTOCOL(serialization, deserialization) accepts bytes for one
         request, decodes according to internal state, pushes
         structured data to handler. accepts structured data f
         handler and encodes and writes to the medium. factory
         handler.
 | structured data
 V
HANDLER (domain logic) accepts structured data, operates state
         machine until the request can be satisfied,
         sends structured data to the protocol.
4
```

Request handlers are registered in the *bzrlib.smart.request* module.

## **Client-side**

Sketch:

```
CLIENT
        domain logic, accepts domain requests, generated struc
        data, reads structured data from responses and turns i
        domain data. Sends structured data to the protocol.
        Operates state machines until the request can be deliv
         (e.g. reading from a bundle generated in bzrlib to del
        complete request).
        This is RemoteBzrDir, RemoteRepository, etc.
Λ
 | structured data
PROTOCOL (serialization, deserialization) accepts structured
         request, encodes and writes to the medium. Reads byte
        medium, decodes and allows the client to read structur
 Λ
 | bytes.
V
```

MEDIUM	accepts	bytes	from	the	proto	ocol &	deli	vers	to	the	remo
	Allows	the pro	otoco]	L to	read	bytes	e.g.	sock	et,	рір	е, н
<b>↓</b> ]											•

The domain logic is in *bzrlib.remote*: *RemoteBzrDir*, *RemoteBranch*, and so on.

There is also an plain file-level transport that calls remote methods to manipulate files on the server in *bzrlib.transport.remote*.

## **Protocol description**

#### Version one

Version one of the protocol was introduced in Bazaar 0.11.

The protocol (for both requests and responses) is described by:

```
REQUEST := MESSAGE_V1
RESPONSE := MESSAGE_V1
MESSAGE_V1 := ARGS [BODY]
ARGS := ARG [MORE_ARGS] NEWLINE
MORE_ARGS := SEP ARG [MORE_ARGS]
SEP := 0x01
BODY := LENGTH NEWLINE BODY_BYTES TRAILER
LENGTH := decimal integer
TRAILER := "done" NEWLINE
```

That is, a tuple of arguments separated by Ctrl-A and terminated with a newline, followed by length prefixed body with a constant trailer. Note that although arguments are not 8-bit safe (they cannot include 0x01 or 0x0a bytes without breaking the protocol encoding), the body is.

#### Version two

Version two was introduced in Bazaar 0.16.

The request protocol is:

REQUEST\_V2 := "bzr request 2" NEWLINE MESSAGE\_V2

The response protocol is:

```
RESPONSE_V2 := "bzr response 2" NEWLINE RESPONSE_STATUS NEWLINE
RESPONSE_STATUS := "success" | "failed"
```

Future versions should follow this structure, like version two does:

FUTURE\_MESSAGE := VERSION\_STRING NEWLINE REST\_OF\_MESSAGE

This is so that clients and servers can read bytes up to the first newline byte to determine what version a message is.

For compatibility will all versions (past and future) of bzr clients, servers that receive a request in an unknown protocol version should respond with a single-line error terminated with 0x0a (NEWLINE), rather than structured response prefixed with a version string.

Version two of the message protocol is:

```
MESSAGE_V2 := ARGS [BODY_V2]
BODY_V2 := BODY | STREAMED_BODY
```

That is, a version one length-prefixed body, or a version two streamed body.

#### Version two with streamed bodies

An extension to version two allows streamed bodies. A streamed body looks a lot like HTTP's chunked encoding:

```
STREAMED_BODY := "chunked" NEWLINE CHUNKS TERMINATOR
CHUNKS := CHUNK [CHUNKS]
CHUNK := HEX_LENGTH CHUNK_CONTENT
HEX_LENGTH := HEX_DIGITS NEWLINE
CHUNK_CONTENT := bytes
TERMINATOR := SUCCESS_TERMINATOR | ERROR_TERMINATOR
SUCCESS_TERMINATOR := 'END' NEWLINE
ERROR_TERMINATOR := 'ERR' NEWLINE CHUNKS SUCCESS_TERMINATOR
```
That is, the body consists of a series of chunks. Each chunk starts with a length prefix in hexadecimal digits, followed by an ASCII newline byte. The end of the body is signaled by 'END\\n', or by 'ERR\\n' followed by error args, one per chunk. Note that these args are 8-bit safe, unlike request args.

A streamed body starts with the string "chunked" so that legacy clients and servers will not mistake the first chunk as the start of a version one body.

The type of body (length-prefixed or chunked) in a response is always the same for a given request method. Only new request methods introduced in Bazaar 0.91 and later use streamed bodies.

### Version three

**Note:** For some discussion of the requirements that led to this new protocol version, see bug #83935.

Version three has bencoding of most protocol structures, to make parsing simpler. For extra parsing convenience, these structures are length prefixed:

LENGTH\_PREFIX := 32-bit unsigned integer in network byte order

Unlike earlier versions, clients and servers are no longer required to know which request verbs and responses will have bodies attached. Because of length-prefixing and other changes, it is always possible to know when a complete request or response has been read, even if the server implements no verbs.

The underlying message format is:

```
MESSAGE := MAGIC NEWLINE HEADERS CONTENTS END_MESSAGE
MAGIC := "bzr message 3 (bzr 1.6)"
HEADERS := LENGTH_PREFIX bencoded_dict
```

```
END_MESSAGE := "e"
BODY := MESSAGE_PART+
MESSAGE_PART := ONE_BYTE | STRUCTURE | BYTES
ONE_BYTE := "o" byte
STRUCTURE := "s" LENGTH_PREFIX bencoded_structure
BYTES := "b" LENGTH_PREFIX bytes
```

(Where + indicates one or more.)

This format allows an arbitrary sequence of message parts to be encoded in a single message. The contents of a MESSAGE have a higher-level message, but knowing just this amount of data it's possible to deserialize and consume a message, so that implementations can respond to messages sent by later versions.

#### Headers

Each request and response will have "headers", a dictionary of keyvalue pairs. The keys must be strings, not any other type of value.

Currently, the only defined header is "Software version". Both the client and the server should include a "Software version" header, with a value of a free-form string such as "bzrlib 1.5", to aid debugging and logging. Clients and servers **should not** vary behaviour based on this string.

#### Conventional requests and responses

By convention, most requests and responses have a simple "arguments plus optional body" structure, as in earlier protocol versions. This section describes how such messages are encoded. All requests and responses defined by earlier protocol versions must be encoded in this way.

Conventional requests will send a CONTENTS of

```
CONV_REQ := ARGS SINGLE_OR_STREAMED_BODY?
SINGLE_OR_STREAMED_BODY := BYTES
                                   BYTES+ TRAILER
ARGS := STRUCTURE(argument_tuple)
TRAILER := SUCCESS_STATUS | ERROR
SUCCESS_STATUS := ONE_BYTE("S")
ERROR := ONE_BYTE("E") STRUCTURE(argument_tuple)
```

Conventional responses will send CONTENTS of

```
CONV_RESP := RESP_STATUS ARGS SINGLE_OR_STREAMED_BODY?
RESP_STATUS := ONE_BYTE("S") | ONE_BYTE("E")
```

If the RESP\_STATUS is success ("S"), the arguments are the method-dependent result.

For errors (where the Status byte of a response or a streamed body is "E"), the situation is analagous to requests. The first item in the encoded sequence must be a string of the error name. The other arguments supply details about the error, and their number and types will depend on the type of error (as identified by the error name).

Note that the streamed body from version two is now just multiple BYTES parts.

The end of the request or response is indicated by the lower-level END\_MESSAGE. If there's only one BYTES element in the body, the TRAILER may or may not be present, depending on whether it was sent as a single chunk or as a stream that happens to have one element.

*(Discussion)* The success marker at the end of a streamed body seems redundant; it doesn't have space for any arguments, and the end of the body is marked anyhow by the end of the message. Recipients shouldn't take any action on it, though they should map an error into raising an error locally.

1.10 clients don't assert that they get a status byte at the end of the message. They will complain (in ConventionalResponseHandler.byte\_part\_received) if they get an initial success and then another byte part with no intervening bytes. If we stop sending the final success message and only flag errors they'll only get one if the error is detected after streaming starts but before any bytes are actually sent. Possibly we should wait until at least the first chunk is ready before declaring success.

For new methods, these sequences are just a convention and may be varied if appropriate for a particular request or response. However, each request should at least start with a STRUCTURE encoding the arguments tuple. The first element of that tuple must be a string that names the request method. (Note that arguments in this protocol version are bencoded. As a result, unlike previous protocol versions, arguments in this version are 8-bit clean.)

(Discussion) We're discussing having the byte segments be not just a method for sending a stream across the network, but actually having them be preserved in the rpc from end to end. This may be useful when there's an iterator on one side feeding in to an iterator on the other, if it avoids doing chunking and byte-counting at two levels, and if those iterators are a natural get granularity. Also, place to good for cases like insert record stream the server can't do much with the data until it gets a whole chunk, and so it'll be natural and efficient for it to be called with one chunk at a time.

On the other hand, there may be times when we've got some bytes from the network but not a full chunk, and it might be worthwhile to pass it up. If we promise to preserve chunks, then to do this we'd need two separate streaming interfaces: "we got a chunk" and "we got some bytes but not yet a full chunk". For insert\_record\_stream the second might not be useful, but it might be good when writing to a file where any number of bytes can be processed.

If we promise to preserve chunks, it'll tend to make some RPCs work only in chunks, and others just on whole blocks, and we can't so easily migrate RPCs from one to the other transparently to older implementations.

The data inside those chunks will be serialized anyhow, and possibly the data inside them will already be able to be serialized apart without understanding the chunks. Also, we might want to use these formats e.g. for pack files or in bundles, and so they don't particularly need lower-level chunking. So the current (unmerged, unstable) record stream serialization turns each record into a bencoded tuple and it'd be feasible to parse one tuple at a time from a byte stream that contains a sequence of them.

So we've decided that the chunks won't be semantic, and code should not count on them being preserved from client to server.

#### Early error returns

(*Discussion*) It would be nice if the server could notify the client of errors even before a streaming request has finished. This could cover situtaions such as the server not understanding the request, it being unable to open the requested location, or it finding that some of the revisions being sent are not actually needed.

Especially in the last case, we'd like to be able to gracefully notice the condition while the client is writing, and then have it adapt its behaviour. In any case, we don't want to have drop and restart the network stream. It should be possible for the client to finish its current chunk and then its message, possibly with an error to cancel what's already been sent.

This relies on the client being able to read back from the server while it's writing. This is technically difficult for http but feasible over a socket or ssh.

We'd need a clean way to pass this back to the request method, even though it's presumably in the middle of doing its body iterator. Possibly the body iterator could be manually given a reference to the request object, and it can poll it to see if there's a response.

Perhaps we need to distinguish error conditions, which should turn into a client-side error regardless of the request code, from early success, which should be handled only if the request code specifically wants to do it.

#### Full-duplex operation

Code not geared to do pipelined requests, and this might require doing asynchrony within bzrlib. We might want to either go fully pipelined and asynchronous, but there might be a profitable middle ground.

The particular case where duplex communication would be good is in working towards the common points in the graphs between the client and server: we want to send speculatively, but detect as soon as they've matched up.

So we could for instance have a synchronous core, but rely on the OS network buffering to allow us to work on batches of say 64kB. We can also pipeline requests and responses, without allowing for them happening out of order, or mixed requests happening at the same time.

Wonder how our network performance would have turned out now if we'd done full-duplex from the start, and ignored hpss over http. We have pretty good (readonly) http support just over dumb http, and that may be better for many users.

## APIs

On the client, the bzrlib code is "in charge": when it makes a request, or asks from data from the network, that causes network IO. The server is event driven: the network code tells the response handler when data has been received, and it takes back a Response object from the request handler that is then polled for body stream data.

### Paths

Paths are passed across the network. The client needs to see a namespace that includes any repository that might need to be referenced, and the client needs to know about a root directory beyond which it cannot ascend.

Servers run over ssh will typically want to be able to access any path the user can access. Public servers on the other hand (which might be over http, ssh or tcp) will typically want to restrict access to only a particular directory and its children, so will want to do a software virtual root at that level. In other words they'll want to rewrite incoming paths to be under that level (and prevent escaping using ../ tricks). The default implementation in bzrlib does this using the *bzrlib.transport.chroot* module.

URLs that include ~ are passed across to the server verbatim and the server can expand them. The default implementation in bzrlib does this using *bzrlib.transport.pathfilter* and *os.path.expanduser*, taking care to respect the virtual root.

Paths in request arguments are UTF-8 encoded, except for the legacy VFS requests which expect escaped (*bzrlib.urlutils.escape*) paths.

# Requests

The first argument of a request specifies the request method.

The available request methods are registered in *bzrlib.smart.request*.

**XXX**: ideally the request methods should be documented here. Contributions welcome!

## **Recognised errors**

The first argument of an error response specifies the error type.

One possible error name is UnknownMethod, which means the server does not recognise the verb used by the client's request. This error was introduced in version three.

**XXX**: ideally the error types should be documented here. Contributions welcome!

igstyle  Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Specifications »	

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

# Plugin API

Date: 2009-01-23

#### Contents

- Plugin API
  - Introduction
    - See also
  - Structure of a plugin
  - Plugin metadata before installation
    - Metadata protocol
    - Control Formats
    - Example
  - Plugin metadata after installation
    - Help and documentation
    - API version
    - Plugin version
    - Detecting whether code's being loaded as a plugin
  - Plugin performance
  - Plugin registrations
  - Publishing your plugin

# Introduction

bzrlib has a very flexible internal structure allowing plugins for many operations. Plugins can add commands, new storage formats, diff and merge features and more. This document provides an overview of the API and conventions for plugin authors.

If you're writing a plugin and have questions not addressed by this document, please ask us.

### See also

- Bazaar Developer Documentation Catalog.
- Bazaar Plugins Guide for more suggestions about particular APIs.

# Structure of a plugin

Plugins are Python modules under <a href="bzrlib.plugins">bzrlib.plugins</a>. They can be installed either into the PYTHONPATH in that location, or in ~/.bazaar/plugins.

Plugins should have a setup.py.

As for other Python modules, the name of the directory must match the expected name of the plugin.

# Plugin metadata before installation

Plugins can export a summary of what they provide, and what versions of bzrlib they are compatible with. This allows tools to be written to work with plugins, such as to generate a directory of plugins, or install them via a symlink/checkout to ~/.bazaar/plugins.

This interface allows bzr to interrogate a plugin without actually loading it. This is useful because loading a plugin may have side effects such as registering or overriding commands, or the plugin may raise an error, if for example a prerequisite is not present.

### Metadata protocol

A plugin that supports the bzr plugin metadata protocol will do two things. Firstly, the setup.py for the plugin will guard the call to setup():

```
if __name__ == 'main':
    setup(...)
```

Secondly, the setup module will have one or more of the following variables present at module scope. Any variables that are missing will be given the defaults from the table. An example of every variable is provided after the full list.

Variable	Default Definition	
bzr_plugin_name	None	The name the plugin package should be given on disk. The plugin is then available to python at bzrlib.plugins.NAME
		A list of the commands that the plugin provides. Commands that already exist in bzr and are

bzr_commands	0	decorated by the plugin do not need to be listed (but it is not harmful if you do list them).
bzr_plugin_version	None	A version_info 5-tuple with the plugins version.
bzr_minimum_version	None	A version_info 3-tuple for comparison with the bzrlib minimum and current version, for determining likely compatibility.
bzr_maximum_version	None	A version_info 3-tuple like bzr_minimum_version but checking the upper limits supported.
bzr_control_formats	8	A dictionary of descriptions of version control directories. See <i>Control Formats</i> below.
bzr_checkout_formats	{}	A dictionary of tree_format_string - > human description strings, for tree formats that drop into the .bzr/checkout metadir system.
bzr_branch_formats	{}	As bzr_checkout_formats but for branches.
bzr_repository_formats	{}	As bzr_checkout_formats but for repositories.
bzr_transports	0	URL prefixes for which this plugin will register transports.

### **Control Formats**

Because disk format detection for formats that bzr does not understand at all can be useful, we allow a declarative description of the shape of a control directory. Each description has a name for showing to users, and a dictonary of relative paths, and the content needed at each path. Paths that end in '/' are required to be directories and the value for that key is ignored. Other paths are required to be regular files, and the value for that key is either None, in which case the file is statted but the content is ignored, or a literal string which is compared against for the content of the file. Thus:

```
# (look for a .hg directory)
bzr_control_formats = {"Mercurial":{'.hg/': None}}
# (look for a file called .svn/format with contents 4\n).
bzr_control_formats = {"Subversion":{'.svn/format': '4\n'}}
```

### Example

An example setup.py follows:

```
#!/usr/bin/env python2.4
from distutils.core import setup
bzr_plugin_name = 'demo'
bzr_commands = [
    'new-command',
    1
bzr branch formats = {
    "Branch label on disk\n":"demo branch",
    }
bzr_control_formats = {"Subversion":{'.svn/format': '4\n'}}
bzr_transports = ["hg+ssh://"]
bzr_plugin_version = (1, 3, 0, 'dev', 0)
bzr_minimum_version = (1, 0, 0)
if __name__ == 'main':
    setup(name="Demo",
          version="1.3.0dev0",
          description="Demo plugin for plugin metadata.",
          author="Canonical Ltd",
          author_email="bazaar@lists.canonical.com",
          license = "GNU GPL v2",
          url="https://launchpad.net/bzr-demo",
          packages=['bzrlib.plugins.demo',
                     'bzrlib.plugins.demo.tests',
```

],
package\_dir={'bzrlib.plugins.demo': '.'})

### Plugin metadata after installation

After a plugin has been installed, metadata can be more easily obtained by looking inside the module object – in other words, for variables defined in the plugin's \_\_init\_\_.py.

### Help and documentation

The module docstring is used as the plugin description shown by bzr plugins. As with all Python docstrings, the first line should be a short complete sentence summarizing the plugin. The full docstring is shown by bzr help PLUGIN\_NAME.

Remember that to be effective, the module docstring must be the first statement in the file. It may come after comments but it must be before any import statements.

### **API version**

Plugins can and should declare that they depend on a particular version of bzrlib like so:

```
from bzrlib.api import require_api
```

```
require_api(bzrlib, (1, 11, 0))
```

Please see API versioning for more details on the API metadata protocol used by bzrlib.

### **Plugin version**

The plugin should expose a version tuple to describe its own version. Some plugins use a version number that corresponds to the version of bzr they're released against, but you can use whatever you want. For example:

```
version_info = (1, 10, 0)
```

Detecting whether code's being loaded as a plugin

You may have a Python module that can be used as a bzr plugin and also in other places. To detect whether the module is being loaded by bzr, use something like this:

```
if ___name__ == 'bzrlib.plugins.loggerhead':
    # register with bzrlib...
```

# Plugin performance

Plugins should avoid doing work or loading code from the plugin or external libraries, if they're just installed but not actually active, because this slows down every invocation of bzr. The bzrlib APIs generally allow the plugin to 'lazily' register methods to invoke if a particular disk format or seen or a particular command is run.

# **Plugin registrations**

The plugin <u>\_\_init\_\_</u>.py runs when the plugin is loaded during bzr startup. Generally the plugin won't want to actually do anything at this time other than register or override functions to be called later.

The plugin can import bzrlib and call any function. Some interesting APIs are described in Bazaar Plugins Guide.

# Publishing your plugin

When your plugin is basically working you might like to share it with other people. Here are some steps to consider:

- make a project on Launchpad.net like <https://launchpad.net/bzr-fastimport> and publish the branches or tarballs there
- include the plugin in <a href="http://wiki.bazaar.canonical.com/BzrPlugins">http://wiki.bazaar.canonical.com/BzrPlugins</a>
- post about it to the bazaar-announce list at lists.canonical.com

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next Specifications »

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »



### Status

Date: 2007-07-08

This document describes the services repositories offer and need to offer within bzrlib.

#### Contents Repositories • Status Motivation • Terminology Command Requirements Data access patterns Patterns used Facilities to scale well Indices Index size Index ordering Changing our current indexes Replace .kndx Data Moving to pack based repositories Naming of files Discovery of files Housing files Combining indices on demand Merging data on push Choosing compression/delta support • Caching and writeing of data Locks Write Groups

# **Motivation**

To provide clarity to API and performance tradeoff decisions by centralising the requirements placed upon repositories.

# Terminology

A **repository** is a store of historical data for bzr.

# **Command Requirements**

Command	Needed services
Add	None
Annotate	Annotated file texts, revision details
Branch	Fetch, Revision parents, Inventory contents, All file texts
Bundle	Maximally compact diffs (file and inventory), Revision graph difference, Revision texts.
Commit	Insert new texts, insert new inventory via delta, insert revision, insert signature
Fetching	Revision graph difference, ghost identification, stream data introduced by a set of revisions in some cheap form, insert data from a stream, validate data during insertion.
Garbage Collection	Exclusive lock the repository preventing readers.
Revert	Delta from working tree to historical tree, and then arbitrary file access to obtain the texts of differing files.
Uncommit	Revision graph access.
Status	Revision graph access, revision text access, file fingerprint information, inventory differencing.
Diff	As status but also file text access.
Merge	As diff but needs up to twice as many file texts - base and other for each changed file. Also an initial fetch is needed.
Log	Revision graph (entire at the moment) access, sometimes status between adjacent revisions. Log of a file needs per-file-graph. Dominator caching or similar tools may be needed to prevent entire graph access.
Missing	Revision graph access, and revision texts to show output.
Update	As for merge, but twice.

### Data access patterns

Ideally we can make our data access for commands such as branch to dovetail well with the native storage in the repository, in the common case. Doing this may require choosing the behaviour of some commands to allow us to have a smaller range of access patterns which we can optimise more heavily. Alternatively if each command is very predicable in its data access pattern we may be able to hint to the low level layers which pattern is needed on a per command basis to get efficient behaviour.

Command	Data access pattern
Annotate- cached	Find text name in an inventory, Recreate one text, recreate annotation regions
Annotate-on demand	Find file id from name, then breadth-first pre-order traversal of versions-of-the-file until the annotation is complete.
Branch	Fetch, possibly taking a copy of any file present in a nominated revision when it is validated during fetch.
Bundle	Revision-graph as for fetch; then inventories for selected revision_ids to determine file texts, then mp-parent deltas for all determined file texts.
Commit	Something like basis-inventories read to determine per-file graphs, insertion of new texts (which may be delta compressed), generation of annotation regions if the repository is configured to do so, finalisation of the inventory pointing at all the new texts and finally a revision and possibly signature.
Fetching	Revision-graph searching to find the graph difference. Scan the inventory data introduced during the selected revisions, and grab the on disk data for the found file texts, annotation region data, per-file-graph data, piling all this into a

	stream.
Garbage Collection	Basically a mass fetch of all the revisions which branches point at, then a bait and switch with the old repository thus removing unreferenced data.
Revert	Revision graph access for the revision being reverted to, inventory extraction of that revision, dirblock-order file text extract for files that were different.
Uncommit	Revision graph access to synthesise pending- merges linear access down left-hand-side, with is_ancestor checks between all the found non- left-hand-side parents.
Status	Lookup the revisions added by pending merges and their commit messages. Then an inventory difference between the trees involved, which may include a working tree. If there is a working tree involved then the file fingerprint for cache-misses on files will be needed. Note that dirstate caches most of this making repository performance largely irrelevant: but if it was fast enough dirstate might be able to be simpler/
Diff	As status but also file text access for every file that is different - either one text (working tree diff) or a diff of two (revision to revision diff).
Merge	As diff but needs up to twice as many file texts - base and other for each changed file. Also an initial fetch is needed. Note that the access pattern is probably id-based at the moment, but that may be 'fixed' with the iter_changes based merge. Also note that while the texts from OTHER are the ones accessed, this is equivalent to the <b>newest</b> form of each text changed from BASE to OTHER. And as the repository looks at when data is introduced, this should be the pattern we focus on for merge.
Log	Revision graph (entire at the moment) access, log of a file wants a per-file-graph. Log -v will want newest-first inventory deltas between revisions.

Missing	Revision graph access, breadth-first pre-order.
Update	As for merge, but twice.

### Patterns used

Note that these are able to be changed by changing what we store. For instance if the repository satisfies mpdiff requests, then bundle can be defined in terms of mpdiff lookups rather than file text lookups appropriate to create mpdiffs. If the repository satisfies full text requests only, then you need the topological access to build up the desired mpdiffs.

Pattern	Commands
Single file text	annotate, diff
Files present in one revision	branch
Newest form of files altered by revisions	merge, update?
Topological access to file versions/deltas	annotate-uncached
Stream all data required to recreate revs	branch (lightweight)
Stream file texts in topological order	bundle
Write full versions of files, inv, rev, sig	commit
Write deltas of files, inv for one tree	commit
Stream all data introduced by revs	fetch
Regenerate/combine deltas of many trees	fetch, pack
Reconstruct all texts and validate trees	check, fetch
Revision graph walk	fetch, pack, uncommit, annotate-uncached, merge, log, missing
Top down access multiple invs concurrently	status, diff, merge?, update?
Concurrent access to N file texts	diff, merge
Iteration of inventory deltas	log -v, fetch?

## Facilities to scale well

### Indices

We want < linear access to all data in the repository. This suggests everything is indexed to some degree.

Often we know the kind of data we are accessing; which allows us to partition our indices if that will help (e.g. by reducing the total index size for queries that only care about the revision graph).

Indices that support our data access patterns will usually display increased locality of reference, reducing the impact of a large indices without needing careful page size management or other tricks.

We need repository wide indices. For the current repositories this is achieved by dividing the keyspace (revisions, signatures, inventories, per-fileid) and then having an append only index within each keyspace. For pack based repositories we will want some means to query the index of each component pack, presumably as a single logical index.

It would be nice if indexing was made cleanly separate from storage. So that suggests indices don't know the meaning of the lookup; indices which offer particular ordering, or graph walking facilities will clearly need that information, but perhaps they don't need to know the semantics ?

#### Index size

Smaller indexes are good. We could go with one big index, or a different index for different operation styles. As multiple indices will occupy more space in total we should consider carefully about adding indices.

#### Index ordering

Looking at the data access patterns some operations such as graph walking can clearly be made more efficient by offering direct iteration rather than repeated reentry into the index - so having indices that support iteration in such a style would be useful eventually.

#### Changing our current indexes

We can consider introducing cleaner indices in advance of a full pack based repository.

There are many possibilities for this, but I've chosen one that seems ok to me for illustration.

A key element is to consider when indices are updated. I think that the update style proposed for pack based repositories - write once, then when we group data again rewrite a new single index - is sufficent.

#### Replace .kndx

We could discard the per-knit .kndx by writing a new index at the end of every bzr transaction indexing the new data introduced by the bzr operation. e.g. at the end of fetch. This can be based on the new GraphIndex index type.

Encoding a knit entry into a GraphIndex can be done as follows:

- Change the key to include a prefix of the knit name, to allow filtering out of data from different knits.
- Encode the parents from the knit as the zeroth node reference list.
- If the knit hunk was delta compressed encode the node it was delta compressed against as the 1st node reference list
(otherwise the 1st node reference list will be empty to indicate no compression parents).

• For the value encode similarly to the current knit format the byte offset for the data record in the knit, the byte length for the data record in the knit and the no-end-of-line flag.

Its important to note that knit repositories cannot be regenerated by scanning .knits, so a mapped index is still irreplaceable and must be transmitted on push/pull.

A potential improvement exists by specialising this further to not record data that is not needed - e.g. an index of revisions does not need to support a pointer to a parent compressed text as revisions.knit is not delta-compressed ever. Likewise signatures do not need the parent pointers at all as there is no 'signature graph'.

### Data

### Moving to pack based repositories

We have a number of challenges to solve.

### Naming of files

As long as the file name is unique it does not really matter. It might be interesting to have it be deterministic based on content, but there are no specific problems we have solved by doing that, and doing so would require hashing the full file. OTOH hashing the full file is a cheap way to detect bit-errors in transfer (such as windows corruption). Non-reused file names are required for data integrity, as clients having read an index will readv at arbitrary times later.

#### Discovery of files

With non-listable transports how should the collection of pack/index

files be found ? Initially record a list of all the pack/index files from write actions. (Require writable transports to be listable). We can then use a heuristic to statically combine pack/index files later.

Housing files

Combining indices on demand

Merging data on push

A trivial implementation would be to make a pack which has just the data needed for the push, then send that. More sophisticated things would be streaming single-pass creation, and also using this as an opportunity to increase the packedness of the local repo.

Choosing compression/delta support

## Caching and writeing of data

Repositories try to provide a consistent view of the data within them within a 'lock context'.

### Locks

Locks come in two flavours - read locks and write locks. Read locks allow data to be read from the repository. Write locks allow data to be read and signal that you intend to write data at some point. The actual writing of data must take place within a Write Group.

Write locks provide a cache of repository data during the period of the write lock, and allow write\_groups to be acquired. For some repositories the presence of a write lock is exclusive to a single client, for others which are lock free or use server side locks (e.g. svn), the write lock simply provides the cache context.

### Write Groups

Write groups are the only allowed means for inserting data into a repository. These are created by start\_write\_group, and concluded by either commit\_write\_group Or abort\_write\_group. A write lock must be held on the repository for the entire duration. At most one write group can be active on a repository at a time.

Write groups signal to the repository the window during which data is actively being inserted. Several write groups could be committed during a single lock.

There is no guarantee that data inserted during a write group will be invisible in the repository if the write group is not committed. Specifically repositories without atomic insertion facilities will be writing data as it is inserted within the write group, and may not be able to revert that data - e.g. in the event of a dropped SFTP connection in a knit repository, inserted file data will be visible in the repository. Some repositories have an atomic insertion facility, and for those all-or-nothing will apply.

The precise meaning of a write group is format specific. For instance a knit based repository treats the write group methods as dummy calls, simply meeting the api that clients will use. A pack based repository will open a new pack container at the start of a write group, and rename it into place at commit time.

◆ Home   Documentation   Develo	oper Document Catalog (2.2b1) »	previous   next
Specifications »		

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

## **Repository Streams**

### **Status**

Date: 2008-04-11

This document describes the proposed programming interface for streaming data from and into repositories. This programming interface should allow a single interface for pulling data from and inserting data into a Bazaar repository.

### Contents

- Repository Streams
  - Status
  - Motivation
  - Use Cases
    - Fetch operations
      - Smart server operations
    - Bundles
    - Data conversion
  - Characteristics
    - Single round trip
    - Forward-only reads
  - Serialisation
    - Weaves
    - Bundles
  - Specification
    - Requesting a stream
    - Structure of a stream
    - Consuming a stream
      - factory metadata

## **Motivation**

To eliminate the current requirement that extracting data from a repository requires either using a slow format, or knowing the format of both the source repository and the target repository.

### **Use Cases**

Here's a brief description of use cases this interface is intended to support.

### Fetch operations

We fetch data between repositories as part of push/pull/branch operations. Fetching data is currently an very interactive process with lots of requests. For performance having the data be supplied in a stream will improve push and pull to remote servers. For purely local operations the streaming logic should help reduce memory pressure. In fetch operations we always know the formats of both the source and target.

#### Smart server operations

With the smart server we support one streaming format, but this is only usable when both the client and server have the same model of data, and requires non-optimal IO ordering for pack to pack operations. Ideally we can both provide optimal IO ordering the pack to pack case, and correct ordering for pack to knits.

### Bundles

Bundles also create a stream of data for revisions from a repository. Unlike fetch operations we do not know the format of the target at the time the stream is created. It would be good to be able to treat bundles as frozen branches and repositories, so a serialised stream should be suitable for this.

### Data conversion

At this point we are not trying to integrate data conversion into this interface, though it is likely possible.

## Characteristics

Some key aspects of the described interface are discussed in this section.

### Single round trip

All users of this should be able to create an appropriate stream from a single round trip.

### Forward-only reads

There should be no need to seek in a stream when inserting data from it into a repository. This places an ordering constraint on streams which some repositories do not need.

## Serialisation

At this point serialisation of a repository stream has not been specified. Some considerations to bear in mind about serialisation are worth noting however.

### Weaves

While there shouldn't be too many users of weave repositories anymore, avoiding pathological behaviour when a weave is being read is a good idea. Having the weave itself embedded in the stream is very straight forward and does not need expensive on the fly extraction and re-diffing to take place.

### Bundles

Being able to perform random reads from a repository stream which is a bundle would allow stacking a bundle and a real repository together. This will need the pack container format to be used in such a way that we can avoid reading more data than needed within the pack container's readv interface.

## Specification

This describes the interface for requesting a stream, and the programming interface a stream must provide. Streams that have been serialised should expose the same interface.

### Requesting a stream

To request a stream, three parameters are needed:

- A revision search to select the revisions to include.
- A data ordering flag. There are two values for this -'unordered' and 'topological'. 'unordered' streams are useful when inserting into repositories that have the ability to perform atomic insertions. 'topological' streams are useful when converting data, or when inserting into repositories that cannot perform atomic insertions (such as knit or weave based repositories).
- A complete\_inventory flag. When provided this flag signals the stream generator to include all the data needed to construct the inventory of each revision included in the stream, rather than just deltas. This is useful when converting data from a repository with a different inventory serialisation, as pure deltas would not be able to be reconstructed.

### Structure of a stream

A stream is an object. It can be consistency checked via the check method (which consumes the stream). The iter\_contents method can be used to iterate the contents of the stream. The contents of the stream are a series of top level records, each of which contains one or more bytestrings (potentially as a delta against another item in the repository) and some optional metadata.

### Consuming a stream

To consume a stream, obtain an iterator from the streams iter\_contents method. This iterator will yield the top level records. Each record has two attributes. One is key\_prefix which is a tuple key prefix for the names of each of the bytestrings in the record. The other attribute is entries, an iterator of the individual items in the record. Each item that the iterator yields is a factory which has metadata about the entry and the ability to return the compressed bytes. This factory can be decorated to allow obtaining different representations (for example from a compressed knit fulltext to a plain fulltext).

In pseudocode:

```
stream = repository.get_repository_stream(search, UNORDERED, Fa
for record in stream.iter_contents():
    for factory in record.entries:
        compression = factory.storage_kind
        print "Object %s, compression type %s, %d bytes long."
        record.key_prefix + factory.key,
        compression, len(factory.get_bytes_as(compression))
```

This structure should allow stream adapters to be written which can coerce all records to the type of compression that a particular client needs. For instance, inserting into weaves requires fulltexts, so a stream would be adapted for weaves by an adapter that takes a stream, and the target weave, and then uses the target weave to reconstruct full texts (which is all that the weave inserter would ask for). In a similar approach, a stream could internally delta compress many fulltexts and be able to answer both fulltext and compressed record requests without extra IO.

#### factory metadata

Valid attributes on the factory are:

- sha1: Optional ascii representation of the sha1 of the bytestring (after delta reconstruction).
- storage\_kind: Required kind of storage compression that has been used on the bytestring. One of mpdiff, knit-annotatedft, knit-annotated-delta, knit-ft, knit-delta, fulltext.
- parents: Required graph parents to associate with this bytestring.
- compressor\_data: Required opaque data relevant to the storage\_kind. (This is set to None when the compressor has no special state needed)
- key: The key for this bytestring. Like each parent this is a tuple that should have the key\_prefix prepended to it to give the unified repository key name.

Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Specifications »	

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

## Case Insensitive File Systems

Bazaar must be portable across operating-systems and file-systems. While the primary file-system for an operating-system might have some particular characteristics, it's not necessary that *all* file-systems for that operating-system will have the same characteristics.

For example, the FAT32 file-system is most commonly found on Windows operating systems, and has the characteristics usually associated with a Windows file-system. However, USB devices means FAT32 file-systems are often used with Linux, so the current operating system doesn't necessarily reflect the capabilities of the file-system.

Bazaar supports 3 kinds of file-systems, each to different degrees.

- Case-sensitive file-systems: This is the file-system generally used on Linux 2 files can differ only by case, and the exact case must be used when opening a file.
- Case-insensitive, case-preserving (cicp) file-systems: This is the file-system generally used on Windows; FAT32 is an example of such a file-system. Although existing files can be opened using any case, the exact case used to create the file is preserved and available for programs to query. Two files that differ only by case is not allowed.
- Case-insensitive: This is the file-system used by very old Windows versions and is rarely encountered "in the wild". Two files that differ only by case is not allowed and the case used to create a file is not preserved.

As can be implied by the above descriptions, only the first two are considered relevant to a modern Bazaar.

For more details, including use cases, please see http://bazaar-

vcs.org/CasePreservingWorkingTreeUseCases

## Handling these file-systems

The fundamental problem handling these file-systems is that the user may specify a file name or inventory item with an "incorrect" case - where "incorrect" simply means different than what is stored - from the user's point-of-view, the filename is still correct, as it can be used to open, edit delete etc the item.

The approach Bazaar takes is to "fixup" each of the command-line arguments which refer to a filename or an inventory item - where "fixup" means to adjust the case specified by the user so it exactly matches an existing item.

There are two places this match can be performed against - the filesystem and the Bazaar inventory. When looking at a case-insensitive file-system, it is impossible to have 2 names that differ only by case, so there is no ambiguity. The inventory doesn't have the same rules, but it is expected that projects which wish to work with Windows would, by convention, avoid filenames that differ only by case.

The rules for such fixups turn out to be quite simple:

- If an argument refers to an existing inventory item, we fixup the argument using the inventory. This is, basically, all commands that take a filename or directory argument *other* than 'add' and in some cases 'mv'
- If an argument refers to an existing filename for the creation of an inventory item (eg, add), then the case of the existing file on the disk will be used. However, Bazaar must still check the inventory to prevent accidentally creating 2 inventory items that differ only by case.
- If an argument results in the creation of a *new* filename (eg, a move destination), the argument will be used as specified. Bzr will create a file and inventory item that exactly matches the

case specified (although as above, care must be taken to avoid creating two inventory items that differ only by case.)

## Implementation of support for these filesystems

From the description above, it can be seen the implementation is fairly simple and need not intrude on the internals of Bazaar too much; most of the time it is simply converting a string specified by the user to the "canonical" form as stored in either the inventory or filesystem. These boil down to the following new API functions:

- osutils.canonical\_relpath() like osutils.relpath() but adjust the case of the result to match any existing items.
- Tree.get\_canonical\_inventory\_path somewhat like Tree.get\_symlink\_target(), Tree.get\_file\_by\_path() etc; returns a name with the case adjusted to match existing inventory items.
- osutils.canonical\_relpaths() and Tree.get\_canonical\_inventory\_paths() - like the 'singular' versions above, but accept and return sequences and therefore offer more optimization opportunities when working with multiple names.

The only complication is the requirement that Bazaar not allow the creation of items that differ only by case on such file-systems. For this requirement, case-insensitive and cicp file-systems can be treated the same. The 'case\_sensitive' attribute on a MutableTree is used to control this behaviour.

Home   Documentation	Developer Document Catalog (2.2b1) »	previous   next
Specifications »		

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

## **Development repository formats**

#### Contents

- Development repository formats
  - Using development repository formats
    - Motivation
    - Format names
    - Support for upgrade and migration
    - Before converting to a development format
    - Creating a new development format branch
    - Creating a new development format repository
    - Upgrading an existing branch or repository to development
    - Starting a new development format branch from one in an older format
    - Develoment formats for bzr-svn users
    - Reporting problems
  - Technical notes
    - When to create a new development format
    - How to create a new development format
  - Format Details
    - development
    - development-rich-root
    - development-subtree
    - Development6RichRoot[Subtree]

## Using development repository formats

### Motivation

We believe that we can continue to gain substantial performance benefits by altering the repository storage in bzr. The more feedback we can get on the changes during the development process the better.

To make it possible to get more feedback we are going to expose the current development formats to the users of our development trunk 'bzr.dev'. The technical details of the individual formats are at the end of this document.

### Format names

The current development format will be called 'development'. Each time the development format changes, the prior development format will be renamed to e.g. 'development0', 'development1' etc.

When a release of bzr is done, all the older numbered development formats will be removed from 'bzr.dev', so we will not be carrying the code for them around indefinately.

### Support for upgrade and migration

The preservation and renaming policy makes it quite safe for users to test out development formats (though we cannot guarantee bugs of course - it is development code):

• users of a given development format can always get back onto regular formats by switching to the next bzr released version which is guaranteed to be able to upgrade from that development format.

• users that routinely use bzr.dev should upgrade to the most recent development version available before pulling in bzr.dev changes around release time, as that is when old format cleanups will occur.

We cannot guarantee backwards compatability though, because some of the planned work may be 'upgrade only'. Please see bzr help formats for the text of the 'development' format which will indicate its compatability with other formats if you need to interoperate with users or services that do not have bzr.dev.

### Before converting to a development format

Run a bzr check with the version of bzr that you will be using. bzr check gets updated as we find new things that are inconsistent with existing repositories. While only a small number of repositories are likely to have any given error, it is best to check just in case.

If bzr check reports a problem, run this command:

bzr reconcile

Note that reconcile can take many hours, particularly if you are reconciling one of the 'knit' or 'dirstate' format repositories. If you have such a repository, consider upgrading it to 'pack-0.92' first, which will perform reconcile significantly faster.

### Creating a new development format branch

If you're starting a project from scratch, it's easy to make it a development one. Here's how:

In other words, use the normal sequence of commands but add the --development option to the init command.

### Creating a new development format repository

If you're starting a project from scratch and wish to use a shared repository for branches, you can make it a development repository like this:

```
cd my-repo
bzr init-repo --development .
cd my-stuff
bzr init
bzr add
bzr commit -m "initial import"
```

In other words, use the normal sequence of commands but add the --development option to the init-repo command.

# Upgrading an existing branch or repository to development

If you have an existing branch and wish to migrate it to a development format, use the upgrade command like this:

bzr upgrade --development path-to-my-branch

If you are using a shared repository, run:

```
bzr upgrade --development ROOT_OF_REPOSITORY
```

to upgrade the history database. Note that this will not alter the branch format of each branch, so you will need to also upgrade each

branch individually if you are upgrading from an old (e.g. < 0.17) bzr. More modern bzr's will already have the branch format at our latest branch format which adds support for tags.

Starting a new development format branch from one in an older format

This can be done in one of several ways:

- 1. Create a new branch and pull into it
- 2. Create a standalone branch and upgrade its format
- 3. Create a knitpack shared repository and branch into it

Here are the commands for using the pull approach:

```
bzr init --development my-new-branch
cd my-new-branch
bzr pull my-source-branch
```

Here are the commands for using the upgrade approach:

```
bzr branch my-source-branch my-new-branch
cd my-new-branch
bzr upgrade --development .
```

Here are the commands for the shared repository approach:

```
cd my-repo
bzr init-repo --development .
bzr branch my-source-branch my-new-branch
cd my-new-branch
```

As a reminder, any of the above approaches can fail if the source branch has inconsistent data within it and hasn't been reconciled yet. Please be sure to check that before reporting problems.

Develoment formats for bzr-svn users

If you are using bzr-svn or are testing the prototype subtree support, you can still use and assist in testing the development formats. The commands to use are identical to the ones given above except that the name of the format to use is development-subtree.

**WARNING**: Note that bzr only supports one-way conversion **to** the subtree format development-subtree. Once you are using development-subtree you cannot pull or merge back into a regular format such as pack-0.92, development etc.

The development-subtree format is required for the bzr-svn plug-in but should otherwise not be used until the subtree feature is complete within bzr.

### **Reporting problems**

If you need any help or encounter any problems, please contact the developers via the usual ways, i.e. chat to us on IRC or send a message to our mailing list. See http://bazaar-vcs.org/BzrSupport for contact details.

## **Technical notes**

### When to create a new development format

Whenever a code change will result in incorrect behaviour with<br/>existingdevelopmentrepositories.Changesinpush/pull/init/commit/mergehave all been known to do this in the<br/>past.

### How to create a new development format

- 1. Register two new formats with the next available sequence number. e.g. development1 and development1-subtree. (You can see the current development format for an example. These should:
  - Use your new development repository/branch/tree classes
  - Have really bare bones help something like 'changes X to be Y see ...developers/development-repo.html'
  - Be hidden and experimental.
- 2. Change the repository class (or branch or tree) in the development and development-subtree formats to point to the new class you are creating.
- 3. Add a new development format (and tests!). Repository formats are in bzrlib.repofmt. You probably want to reproduce the current development format from bzrlib.repofmt.pack\_repo with just new disk format strings, \_get\_matching\_bzrdir and help.
- 4. Register your development format with the various registries. At the moment you need to update:
  - 1. bzrlib/bzrdir.py to register the WT/Branch/Repository

collection.

- bzrlib/workingtree.py, bzrlib/branch.py, bzrlib/repository.py, each one maintains a direct list of their respective formats.
- 3. For repositories, you also need to update the InterKnit1and2 class. This is responsible for converting between rich-root and non-rich-root repositories.
- For repositories based on KnitPackRepository, you need to update bzrlib/tests/test\_pack\_repository.py to add the class to the tested permutations.
- 5. Alter any other things that do class based tests. The easiest way to find these is a grep for Development in bzrlib and please refactor as you find these to reduce the relevance this step has, as it should not need to exist.
- 6. Now subclass/create from scratch/whatever the live object code you need to change to introduce your new format. Keep in mind that eventually it will become the default format, so please don't keep subclassing the last releases code, rather consider making the last releases code a subclass of your new code (if there is a lot in common) so that we can eventually remove that format once it becomes ancient (or relegate it to a plugin).
- 7. Once you have made the changes that required a new disk format, you should submit the resulting branch to be merged. Other changes - to take advantage of whatever new feature you have added - should be sent in separately, because the disk level changes are a contention point between multiple developers.

### Format Details

### development

Not currently available, as our development formats are all rich root or subtrees now.

### development-rich-root

Currently an alias for Development6Subtree

### development-subtree

Currently an alias for Development6Subtree

### Development6RichRoot[Subtree]

These formats use the new groupcompress delta compress and a CHK(Content Hash Key) based inventory store which is much faster at incremental operations than the prior XML based store. *Note* Converting from a non-rich-root to a rich-root format is a one-way upgrade, and you cannot merge back afterwards: using this format for everyday use is likely to cause all developers of a project to upgrade to a rich-root format themselves. This is fine, as bzr is moving to make rich-root formats the default and to get all users to upgrade, but we have not finalised the migration process, and until we do do not recomment that casual users upgrade. Users of bzr-svn are already using rich-root formats and can test with this with impunity.

Specifications »

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next

Specifications »

## KnitPack repository format

### Contents

- KnitPack repository format
  - Using KnitPack repositories
    - Motivation
    - Preparation
    - Creating a new knitpack branch
    - Creating a new knitpack repository
    - Upgrading an existing branch or repository to knitpack format
    - Starting a new knitpack branch from one in an older format
    - Testing packs for bzr-svn users
    - Reporting problems
  - Technical notes

## Using KnitPack repositories

### Motivation

KnitPack is a new repository format for Bazaar, which is expected to be faster both locally and over the network, is usually more compact, and will work with more FTP servers.

Our benchmarking results to date have been very promising. We fully expect to make a pack-based format the default in the near future. We would therefore like as many people as possible using KnitPack repositories, benchmarking the results and telling us where improvements are still needed.

### Preparation

A small percentage of existing repositories may have some inconsistent data within them. It's is a good idea to check the integrity of your repositories before migrating them to knitpack format. To do this, run:

bzr check

If that reports a problem, run this command:

bzr reconcile

Note that this can take many hours for repositories with deep history so be sure to set aside some time for this if it is required.

Creating a new knitpack branch

If you're starting a project from scratch, it's easy to make it a

knitpack one. Here's how:

cd my-stuff bzr init --pack-0.92 bzr add bzr commit -m "initial import"

In other words, use the normal sequence of commands but add the --pack-0.92 option to the init command.

Note: In bzr 0.92, this format was called knitpack-experimental.

Creating a new knitpack repository

If you're starting a project from scratch and wish to use a shared repository for branches, you can make it a knitpack repository like this:

```
cd my-repo
bzr init-repo --pack-0.92 .
cd my-stuff
bzr init
bzr add
bzr commit -m "initial import"
```

In other words, use the normal sequence of commands but add the --pack-0.92 option to the init-repo command.

### Upgrading an existing branch or repository to knitpack format

If you have an existing branch and wish to migrate it to a knitpack format, use the upgrade command like this:

```
bzr upgrade --pack-0.92 path-to-my-branch
```
If you are using a shared repository, run:

```
bzr upgrade --pack-0.92 ROOT_OF_REPOSITORY
```

to upgrade the history database. Note that this will not alter the branch format of each branch, so you will need to also upgrade each branch individually if you are upgrading from an old (e.g. < 0.17) bzr. More modern bzr's will already have the branch format at our latest branch format which adds support for tags.

# Starting a new knitpack branch from one in an older format

This can be done in one of several ways:

- 1. Create a new branch and pull into it
- 2. Create a standalone branch and upgrade its format
- 3. Create a knitpack shared repository and branch into it

Here are the commands for using the pull approach:

```
bzr init --pack-0.92 my-new-branch
cd my-new-branch
bzr pull my-source-branch
```

Here are the commands for using the upgrade approach:

```
bzr branch my-source-branch my-new-branch
cd my-new-branch
bzr upgrade --pack-0.92 .
```

Here are the commands for the shared repository approach:

```
cd my-repo
bzr init-repo --pack-0.92 .
bzr branch my-source-branch my-new-branch
cd my-new-branch
```

As a reminder, any of the above approaches can fail if the source branch has inconsistent data within it and hasn't been reconciled yet. Please be sure to check that before reporting problems.

#### Testing packs for bzr-svn users

If you are using bzr-svn or are testing the prototype subtree support, you can still use and assist in testing KnitPacks. The commands to use are identical to the ones given above except that the name of the format to use is knitpack-subtree-experimental.

WARNING: Note that the subtree formats, dirstate-subtree and knitpack-subtree-experimental, are **not** production strength yet and may cause unexpected problems. They are required for the bzr-svn plug-in but should otherwise only be used by people happy to live on the bleeding edge. If you are using bzr-svn, you're on the bleeding edge anyway. :-)

#### Reporting problems

If you need any help or encounter any problems, please contact the developers via the usual ways, i.e. chat to us on IRC or send a message to our mailing list. See http://bazaar-vcs.org/BzrSupport for contact details.

#### **Technical notes**

Bazaar 0.92 adds a new format (experimental at first) implemented in bzrlib.repofmt.pack\_repo.py.

This format provides a knit-like interface which is quite compatible with knit format repositories: you can get a VersionedFile for a particular file-id, or for revisions, or for the inventory, even though these do not correspond to single files on disk.

The on-disk format is that the repository directory contains these files and subdirectories:

packs/	completed readonly packs
indices/	indices for completed packs
upload/	temporary files for packs currently being written
obsolete_packs/	packs that have been repacked and are no longer normally needed
pack-names	index of all live packs
lock/	lockdir

Note that for consistency we always write "indices" not "indexes".

This is implemented on top of pack files, which are written once from start to end, then left alone. A pack consists of a body file, plus several index files. There are four index files for each pack, which have the same basename and an extension indicating the purpose of the index:

extn	Purpose	Кеу	References
.tix	File texts	file_id, revision_id	per-file parents, compression basis per- file parents
.six		revision_id,	•

	Signatures		
.rix	Revisions	revision_id,	revision parents
.iix	Inventory	revision_id,	revision parents, compression base

Indices are accessed through the bzrlib.index.GraphIndex class. Indices are stored as sorted files on disk. Each line is one record, and contains:

- key fields
- a value string for all these indices, this is an ascii decimal pair of "offset length" giving the position of the referenced data within the pack body file
- a list of zero or more reference lists

The reference lists let a graph be stored within the index. Each reference list entry points to another entry in the same index. The references are represented as a byte offset for the target within the index file.

When a compression base is given, it indicates that the body of the text or inventory is a forward delta from the referenced revision. The compression base list must have length 0 or 1.

Like packs, indexes are written only once and then unmodified. A GraphIndex builder is a mutable in-memory graph that can be sorted, cross-referenced and written out when the write group completes.

There can also be index entries with a value of 'a' for absent. These records exist just to be pointed to in a graph. This is used, for example, to give the revision-parent pointer when the parent revision is in a previous pack.

The data content for each record is a knit data chunk. The knits are always unannotated - the annotations must be generated when

needed. (We'd like to cache/memoize the annotations.) The data hunks can be moved between packs without needing to recompress them.

It is not possible to regenerate an index from the body file, because it contains information stored in the knit index that's not in the body. (In particular, the per-file graph is only stored in the index.) We would like to change this in a future format.

The lock is a regular LockDir lock. The lock is only held for a much reduced scope, while updating the pack-names file. The bulk of the insertion can be done without the repository locked. This is an implementation detail; the repository user should still call repository.lock\_write at the regular time but be aware this does not correspond to a physical mutex.

Read locks control caching but do not affect writers.

The newly-added repository write group concept is very important to KnitPack repositories. When start\_write\_group is called, a new temporary pack is created and all modifications to the repository will go into it until either commit\_write\_group Or abort\_write\_group is called, at which time it is either finished and moved into place or discarded respectively. Write groups cannot be nested, only one can be underway at a time on a Repository instance and they must occur within a write lock.

Normally the data for each revision will be entirely within a single pack but this is not required.

When a pack is finished, it gets a final name based on the md5 of all the data written into the pack body file.

The pack-names file gives the list of all finished non-obsolete packs. (This should always be the same as the list of files in the packs/

directory, but the file is needed for readonly http clients that can't easily list directories, and it includes other information.) The constraint on the pack-names list is that every file mentioned must exist in the packs/ directory.

In rare cases, when a writer is interrupted, about-to-be-removed packs may still be present in the directory but removed from the list.

As well as the list of names, the pack-names file also contains the size, in bytes, of each of the four indices. This is used to bootstrap bisection search within the indices.

In normal use, one pack will be created for each commit to a repository. This would build up to an inefficient number of files over time, so a repack operation is available to recombine them, by producing larger files containing data on multiple revisions. This can be done manually by running bzr pack, and it also may happen automatically when a write group is committed.

The repacking strategy used at the moment tries to balance not doing too much work during commit with not having too many small files left in the repository. The algorithm is roughly this: the total number of revisions in the repository is expressed as a decimal number, e.g. "532". Then we'll repack until we have five packs containing a hundred revisions each, three packs containing ten revisions each, and two packs with single revisions. This means that each revision will normally initially be created in a single-revision pack, then moved to a ten-revision pack, then to a 100-pack, and so on.

As with other repositories, in normal use data is only inserted. However, in some circumstances we may want to garbage-collect or prune existing data, or reconcile indexes.

🚸 Home	Documentation	Developer Document Catalog (2.2b1) »	previous	next
--------	---------------	--------------------------------------	----------	------

Specifications »

In the second se

## Implementation notes

- BTree Index Prefetch How bzr decides to pre-read extra nodes in the btree index.
- Computing last\_modified values for inventory entries
- Content filtering
- LCA Tree Merging Merging tree-shape when there is not a single unique ancestor (criss-cross merge).

igstyle Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next

Implementation notes »

## **BTree Index Prefetch**

This document outlines how we decide to pre-read extra nodes in the btree index.

### Rationale

Because of the latency involved in making a request, it is often better to make fewer large requests, rather than more small requests, even if some of the extra data will be wasted.

#### Example

Using my connection as an example, I have a max bandwidth of 160kB/s, and a latency of between 100-400ms to London, I'll use 200ms for this example. With this connection, in 200ms you can download 32kB. So if you make 10 requests for 4kB of data, you spend  $10^*.2s = 2s$  sending the requests, and  $4^*10/160 = .25s$  actually downloading the data. If, instead, you made 3 requests for 32kB of data each, you would take  $3^*.2s = .6s$  for requests, and  $32^*3/160 = .6s$  for downloading the data. So you save 2.25 - 1.2 = 1.05s even though you downloaded  $32^*3-4^*10 = 56kB$  of data that you probably don't need. On the other hand, if you made 1 request for 480kB, you would take .2s for the request, and 480/160=3s for the data. So you end up taking 3.2s, because of the wasted 440kB.

#### **BTree Structure**

This is meant to give a basic feeling for how the btree index is laid out on disk, not give a rigorous discussion. For that look elsewhere[ref?].

The basic structure is that we have pages of 4kB. Each page is either a leaf, which holds the final information we are interested in, or is an internal node, which contains a list of references to the next layer of nodes. The layers are structured such that all nodes for the top layer come first, then the nodes for the next layer, linearly in the file.

#### Example 1 layer

In the simplest example, all the data fits into a single page, the root node. This means the root node is a leaf node.

#### Example 2 layer

As soon as the data cannot fit in a single node, we create a new internal node, make that the root, and start to create multiple leaf nodes. The root node then contains the keys which divide the leaf pages. (So if leaf node 1 ends with 'foo' and leaf node 2 starts with 'foz', the root node would hold the key 'foz' at position 0).

#### Example 3 layer

It is possible for enough leaf nodes to be created, that we cannot fit all there references in a single node. In this case, we again split, creating another layer, and setting that as the root. This layer then references the intermediate layer, which references the final leaf nodes. In all cases, the root node is a single page wide. The next layer can have 2-N nodes.

#### Current Info

Empirically, we've found that the number of references that can be stored on a page varies from about 60 to about 180, depending on how much we compress, and how similar the keys are. Internal nodes also achieve approximately the same compression, though they seem to be closer to 80-100 and not as variable. For most of this discussion, we will assume each page holds 100 entries, as that makes the math nice and clean.

So the idea is that if you have <100 keys, they will probably all fit on the root page. If you have 100 - 10,000 keys, we will have a 2-layer structure, if you have 10,000 - 1,000,000 keys, you will have a 3-layer structure. 10^6-10^8 will be 4-layer, etc.

#### Data and Request

It is important to be aware of what sort of data requests will be made on these indexes, so that we know how to optimize them. This is still a work in progress, but generally we are searching through ancestry. The final information (in the leaf nodes) is stored in sorted order. Revision ids are generally of the form "prefix:committer@emailtimestamp-randomtail". This means that revisions made by the same person around the same time will be clustered, but revisions made by different people at the same time will not be clustered. For files, the keys are (file-id, revision-id) tuples. And file-ids are generally basename-timestamp-random-count (depending) on the converter). This means that all revisions for a given file-id will be grouped together, and that files with similar names will be grouped together. However, files committed in the same revisions will not be grouped together in the index.[1]

One interesting possibility would be to change file-ids from being [1] 'basename-...', to being 'containing-dirname-filename-...', which would group files in the similarly named directories together.

In general, we always start with a request for the root node of the index, as it tells us the final structure of the rest of the index. How many total pages, what pages are internal nodes and what layer, which ones are leaves. Before this point, we do know the *size* of the index, because that is stored in the pack-names file.

### Thoughts on expansion

This is just a bullet list of things to consider when expanding a request.

- We generally assume locality of reference. So if we are currently reading page 10, we are more likely to read page 9 or 11 than we are page 20.
- However, locality of reference only really holds within a layer. If we are reading the last node in a layer, we are unlikely to read the first node of the next layer. In fact, we are most likely to read the *last* node of the next layer.

More directly, we are probably equally likely to read any of the nodes in the next layer, which could be referred to by this layer. So if we have a structure of 1 root node, 100 intermediate nodes, and 10,000 leaf nodes. They will have offsets: 0, 1-101, 102-10,102.

If we read the root node, we are likely to want any of the 1-101 nodes (because we don't know where the key points). If we are reading node 90, then we are likely to want a node somewhere around 9,100-9,200.

- When expanding a request, we are considering that we probably want to read on the order of 10 pages extra. (64kB / 4kB = 16 pages.) It is unlikely that we want to expand the requests by 100.
- At the moment, we assume that we don't have an idea of where in the next layer the keys might fall. We *could* use a predictive algorithm assuming homogenous distribution. When reading the root node, we could assume an even distribution from 'a-z', so

that a key starting with 'a' would tend to fall in the first few pages of the next layer, while a key starting with 'z' would fall at the end of the next layer. However, this is quite likely to fail in many ways. Specific examples:

- Converters tend to use an identical prefix. So all revisions will start with 'xxx:', leading us to think that the keys fall in the last half, when in reality they fall evenly distributed.
- When looking in text indexes. In the short term, changes tend to be clustered around a small set of files. Short term changes are unlikely to cross many pages, but it is unclear what happens in the mid-term. Obviously in the long term, changes have happened to all files.

A possibility, would be to use this after reading the root node. And then using an algorithm that compares the keys before and after this record, to find what a distribution would be, and estimate the next pages.

This is a lot of work for a potentially small benefit, though.

- When checking for N keys, we do sequential lookups in each layer. So we look at layer 1 for all N keys, then in layer 2 for all N keys, etc. So our requests will be clustered by layer.
- For projects with large history, we are probably more likely to end up with a bi-modal distribution of pack files. Where we have 1 pack file with a large index, and then several pack files with small indexes, several with tiny indexes, but no pack files with medium sized indexes. This is because a command like bzr pack will combine everything into a single large file. Commands like bzr commit will create an index with a single new record, though these will be packaged together by autopack.

Commands like bzr push and bzr pull will create indexes with more records, but these are unlikely to be a significant portion of the history. Consider bzr has 20,000 revisions, a single push/pull is likely to only be 100-200 revisions, or 1% of the history.

Note that there will always be cases where things are evenly distributed, but we probably shouldn't *optimize* for that case.

- 64kB is 16 pages. 16 pages is approximately 1,600 keys.
- We are considering an index with 1 million keys to be very large. 10M is probably possible, and maybe 100M, but something like 1 billion keys is unlikely. So a 3-layer index is fairly common (it exists already in bzr), but a 4-layer is going to be quite rare, and we will probably never see a 5-layer.
- There are times when the second layer is going to be incompletely filled out. Consider an index with 101 keys. We found that we couldn't fit everything into a single page, so we expanded the btree into a root page and a leaf page, and started a new leaf page. However, the root node only has a single entry. There are 3 pages, but only one of them is "full". This happens again when we get near the 10,000 node barrier. We found we couldn't fit the index in a single page, so we split it into a higher layer, and 1 more sub-layer. So we have 1 root node, 2 layer-2 nodes, and N leaf nodes (layer 3). If we read the first 3 nodes, we will have read all internal nodes.

It is certainly possible to detect this for the first-split case (when things no-longer fit into just the root node), as there will only be a few nodes total. Is it possible to detect this from only the 'size' information for the second-split case (when the index no longer fits in a single page, but still fits in only a small handful of pages)? This only really works for the root + layer 2. For layers 3+ they will always be too big to read all at once. However, until we've read the root, we don't know the layout, so all we have to go on is the size of the index, though that also gives us the explicit total number of pages. So it doesn't help to read the root page and then decide. However, on the flip side, if we read *before* the split, then we don't gain much, as we are reading pages we aren't likely to be interested in.

For example:

We have 100 keys, which fits onto 100 pages, with a single root node. At 1,100 keys, it would be 101 leaf pages, which would then cause us to need 2 index pages, triggering an extra layer. However, this is very sensitive to the number of keys we fit per-page, which depends on the compression. Although, we could consider 2,000 keys. Which would be 200 leaf nodes, and 2 intermediate nodes, and a single root node. It is unlikely that we would ever be able to fit 200 references into a single root node.

So if we pretend that we split at 1 page, 100 pages, and 10,000 pages. We might be able to say, at 1-5 pages, read all pages, for 5-100 pages, read only the root. At 100 - 500 pages, read 1-5 pages, for 500-10,000 read only the root. At 10,000-50,000 read 1-5 pages again, but above 50,000 read only the root. We could bias this a bit smaller, say at powers of 80, instead of powers of 100, etc. The basic idea is that if we are *close* to a layer split, go ahead and read a small number of extra pages.

• The previous discussion applies whenever we have an upper layer that is not completely full. So the pages referenced by the last node from the upper layer will often not have a full 100-way fan out. Probably not worthwhile very often, though. • Sometimes we will be making a very small request for a very small number of keys, we don't really want to bloat tiny requests. Hopefully we can find a decent heuristic to determine when we will be wanting extra nodes later, versus when we expect to find all we want right now.

## Algorithm

This is the basic outline of the algorithm.

- 1. If we don't know the size of the index, don't expand as we don't know what is available. (This only really applies to the packnames file, which is unlikely to ever become larger than 1 page anyway.)
- 2. If a request is already wide enough to be greater than the number of recommended pages, don't bother trying to expand. This only really happens with LocalTransport which recommends a single page.
- 3. Determine what pages have already been read (if any). If the pages left to read can fit in a single request, just request them. This tends to happen on medium sized indexes (ones with low hundreds of revisions), and near the end when we've read most of the whole index already.
- 4. If we haven't read the root node yet, and we can't fit the whole index into a single request, only read the root node. We don't know where the layer boundaries are anyway.
- 5. If we haven't read "tree depth" pages yet, and are only requesting a single new page don't expand. This is meant to handle the 'lookup 1 item in the index' case. In a large pack file, you'll read only a single page at each layer and then be done. When spidering out in a search, this will cause us to take a little bit longer to start expanding, but once we've started we'll be expanding at full velocity. This could be improved by having indexes inform each other that they have already entered the 'search' phase, or by having a hint from above to indicate the same.

However, remember the 'bi-modal' distribution. Most indexes will either be very small, or very large. So either we'll read the whole thing quickly, or we'll end up spending a lot of time in the index. Which makes a small number of extra round trips to large indexes a small overhead. For 2-layer nodes, this only 'wastes' one round trip.

6. Now we are ready to expand the requests. Expand by looking for more pages next to the ones requested that fit within the current layer. If you run into a cached page, or a layer boundary, search further only in the opposite direction. This gives us proper locality of reference, and also helps because when a search goes in a single direction, we will continue to prefetch pages in that direction.

Home   Documentation	Developer Document Catalog (2.2b1) »	previous   next
Implementation notes »		

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next

Implementation notes »

## Computing last\_modified values

### Introduction

Bazaar (through at least 0.19) computes a last\_modified attribute for all inventory entries and stores it at commit time. This is the revision\_id that last changed or merged the file. It is used in knit and weave repositories to look up the file text, and to index into the file graph. It's also used to determine which revisions of the file text to pull during fetch.

This data is not natively stored by most other systems so we need to synthesize it during conversion.

This is a case of non-core data that we might wish to treat as cached, rather than always stored.

## Definition

Take the set of all "heads": all the versions of these files in parent trees.

Reduce the heads by eliminating any whose last\_modified is an ancestor of the last\_modified of any other head.

If there is still more than one head, a new last\_modified is assigned. This points to the merge point in the file graph.

If the file text and properties are the same as the sole remaining head, its last\_modified is inherited. Property changes include executable bit, filename, and containing directory.

Otherwise, a new last\_modified is used.

(This is meant to be the simplest statement, but it may not be the most efficient algorithm; anything that gives equivalent results can be used.)

### Generation in commit

Commit and converters both need this when writing into Bazaar native formats.

This is an O(tree) operation because it needs to check for files with multiple heads. It could be reduced to O(changed\_or\_merged\_files) if that was faster to determine. So it needs to be fast.

For the single-parent commit case, we just need to determine which files have changed compared to the parent. If the file was changed, it gets the revision id of the new revision; otherwise it inherits the value from the parent tree.

In the multi-parent commit case (commit of a merge), it can take the value from any of the parent trees, or of the new revision.

Commit in a dirstate tree should be able to do this more easily by looking at a row of the dirstate to get the per-file parents. It still needs to look at the revision or file graph information to work out whether heads can be eliminated as previously merged. At the moment find previous heads works on inventories, so needs to spend considerable effort building whole inventories, including files that are not modified or merged. (Called from record\_entry\_contents.) It might be better to have the commit builder pass in the per-entry parents so that dirstate can generate that necessary. (See also spec just those are the for iter\_changes\_multiple\_parents.)

If merge used a per-file graph then it would know when one version fully supersedes another, and it could emit only a single parent. Merge could in fact do this even when not using per-file graphs. In the current dirstate format we need to store the full data for all trees because they can be extracted from the dirstate, but it could mark some parents as already merged.

Alternatively, we could change the dirstate to include only the base and current trees, and cache the merged-in parents elsewhere.

(Offtopic other dirstate changes: we could also omit the workingcopy hash, and just have a stat-fingerprint of when it was last known equal to the basis revision. That reduces the amount of data stored and possibly makes it simpler to update, and shouldn't penalize common cases.)

## Generation during conversion

Accessing a foreign branch requires synthesizing this information. If last\_modified is removed from a future bzr version, we will also need to synthesize it to pull back to earlier formats.

Because last\_modified is not natively stored in the foreign branches, we want to take advantage of any conversion we've already done, so that we don't need to recursively generate them on every access. We'd prefer to find a revision that's already converted to a Bazaar inventory within another related repository, such as the target of a conversion.

## Avoiding last\_modified

last\_modified is potentially expensive to determine and we may not want to store it in inventories in future. Therefore we should use it only when necessary:

- When writing out an inventory format that includes it.
- In Bazaar formats that use it as a key for the file text or file ancestry. This should be hidden behind the Repository/RevisionTree interface.
- When a user operation specifically requires the last\_modified (e.g. hypothetical annotate directory).

We already do this in most cases.

Compared to annotate

### Use cases

#### Cases to test

- 1. Single parent, unmodified file
- 2. Single parent, modified file
- 3. Two parents, one descended from the other, modified in one parent only
- 4. Two parents, one descended from the other, modified in one parent only, but also modified locally.
- 5. Two parents, not descended from each other, modified in one parent only.
- 6. Two parents, not descended from each other, modified in one parent only, but also modified locally.
- 7. Two parents, modified in both to different values.
- 8. Two parents, modified in both to the same value.
- 9. Two parents, modified in both, and reverted in both back to the original text.
- 10. Three parents, modified in only one
- 11. Three parents, modified in only one, also modified locally.
- 12. Three parents, modified in 2
- 13. Three parents, modified in 2, and locally.
- 14. Three parents, modified in 2, but one is a descendant of the other.

## Performance considerations

Often we'll want the last\_modified information for multiple files, perhaps everything in a directory or in a whole tree. It may be more efficient for the api to accommodate this. Often the last\_modified will be similar for multiple files, and if we process them all at once we can avoid some repeated work in calculating their heads.

## Open questions

• How does caching find\_heads interact with cherry-picks?

#### Possible structure

For a single file, if I am different from all parents, 'new'. (Do not need to evaluate last modified).

🚸 Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Implementation notes »	

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next

Implementation notes »
# **Content Filtering**

Content filtering is the feature by which Bazaar can do line-ending conversion or keyword expansion so that the files that appear in the working tree are not precisely the same as the files stored in the repository.

This document describes the implementation; see the user guide for how to use it.

We distinguish between the *canonical form* which is stored in the repository and the *convenient form* which is stored in the working tree. The convenient form will for example use OS-local newline conventions or have keywords expanded, and the canonical form will not. We use these names rather than eg "filtered" and "unfiltered" because filters are applied when both reading and writing so those names might cause confusion.

Content filtering is only active on working trees that support it, which is format 2a and later.

Content filtering is configured by rules that match file patterns.

### Filters

Filters come in pairs: a read filter (reading convenient->canonical) and a write filter. There is no requirement that they be symmetric or that they be deterministic from the input, though in general both these properties will be true. Filters are allowed to change the size of the content, and things like line-ending conversion commonly will.

Filters are fed a sequence of byte chunks (so that they don't have to hold the whole file in memory). There is no guarantee that the chunks will be aligned with line endings. Write filters are passed a context object through which they can obtain some information about eg which file they're working on. (See bzrlib.filters docstring.)

These are at the moment strictly *content* filters: they can't make changes to the tree like changing the execute bit, file types, or adding/removing entries.

## Conventions

bzrlib interfaces that aren't explicitly specified to deal with the convenient form should return the canonical form. Whenever we have the SHA1 hash of a file, it's the hash of the canonical form.

## **Dirstate interactions**

The dirstate file should store, in the column for the working copy, the cached hash and size of the canonical form, and the packed stat fingerprint for which that cache is valid. This implies that the stored size will in general be different to the size in the packed stat. (However, it may not always do this correctly - see <https://bugs.edge.launchpad.net/bzr/+bug/418439>.)

The dirstate is given a SHA1Provider instance by its tree. This class can calculate the (canonical) hash and size given a filename. This provides a hook by which the working tree can make sure that when the dirstate needs to get the hash of the file, it takes the filters into account.

## User interface

Most commands that deal with the text of files present the canonical form. Some have options to choose.

## Performance considerations

Content filters can have serious performance implications. For example, getting the size of (the canonical form of) a file is easy and fast when there are no content filters: we simply stat it. However, when there are filters that might change the size of the file, determining the length of the canonical form requires reading in and filtering the whole file.

Formats from 1.14 onwards support content filtering, so having fast paths for the case where content filtering is not possible is not generally worthwhile. In fact, they're probably harmful by causing extra edges in test coverage and performance.

We need to have things be fast even when filters are in use and then possibly do a bit less work when there are no filters configured.

## Future ideas and open issues

- We might benefit from having filters declare some of their properties statically, for example that they're deterministic or can round-trip or won't change the length of the file. However, common cases like crlf conversion are not guaranteed to round-trip and may change the length, so perhaps adding separate cases will just complicate the code and tests. So overall this does not seem worthwhile.
- In a future workingtree format, it might be better not to separately store the working-copy hash and size, but rather just a stat fingerprint at which point it was known to have the same canonical form as the basis tree.
- It may be worthwhile to have a virtual Tree-like object that does filtering, so there's a clean separation of filtering from the ondisk state and the meaning of any object is clear. This would have some risk of bugs where either code holds the wrong object, or their state becomes inconsistent.

This would be useful in allowing you to get a filtered view of a historical tree, eg to export it or diff it. At the moment export needs to have its own code to do the filtering.

The convenient-form tree would talk to disk, and the convenientform tree would sit on top of that and be used by most other bzr code.

If we do this, we'd need to handle the fact that the on-disk tree, which generally deals with all of the IO and generally works entirely in convenient form, would also need to be told the canonical hash to store in the dirstate. This can perhaps be handled by the SHA1Provider or a similar hook. • Content filtering at the moment is a bit specific to on-disk trees: for instance SHA1Provider goes directly to disk, but it seems like this is not necessary.

## See also

- http://bazaar-vcs.org/LineEndings
- http://bazaar-vcs.org/LineEndings/Roadmap
- Developer Documentation
- bzrlib.filters

 $m \rellip$  Home | Documentation | Developer Document Catalog (2.2b1) » -

previous | next

Implementation notes »

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next

Implementation notes »

# LCA Tree Merging

There are 2 ways that you get LCA merge resolution in bzr. First, if you use bzr merge --lca, the *content* of files will be resolved using a Least Common Ancestors algorithm. That is described in <lcamerge.html> not here.

This document describes how we handle merging tree-shape when there is not a single unique ancestor (criss-cross merge). With a single LCA, we use simple 3-way-merge logic.

When there are multiple possible LCAs, we use a different algorithm for handling tree-shape merging. Described here.

As a simple example, here is a revision graph which we will refer to often:

In this graph, THIS and OTHER both have LCA1 and LCA2 in their ancestry but neither is an ancestor of the other, so we have 2 least common ancestors. The unique common ancestor is BASE. (It should be noted that in this text we will talk directly about LCA1 and LCA2, but the algorithms are designed to cope with more than 2 LCAs.)

## Scalars

#### Definition

I'm defining scalar values as ones that cannot be 'merged' on their own. For example, the name of a file is "scalar". If one person changes "foo.txt" to "foo.c" and someone else changes "foo.txt" to "bar.txt" we don't merge the changes to be "bar.c", we simply conflict and expect the user to sort it out.

We use a slightly different algorithm for scalars.

#### **Resolution Algorithm**

(This can be seen as bzrlib.merge.Merge3Merger.\_lca\_multi\_way`

- 1. If THIS and OTHER have the same value, use it. There is no need to inspect any other values in this case. Either nothing was changed (all interesting nodes would have the same value), or we have "accidental convergence" (both sides made the same change.).
- 2. Find the values from LCA1 and LCA2 which are not the same as BASE. The idea here is to provide a rudimentary "heads" comparison. Often, the whole tree graph will have a criss-cross, but the per-file (per-scalar) graph would be linear, and the value in one LCA strictly dominates the other. It is possible to construct a scenario where one side dominates the other, but the dominated value is not BASE, but a second intermediate value. Most scalars are rarely changed, so this is unlikely to be an issue. The trade-off is having to generate and inspect the per-scalar graph.

If there are no LCA values that are different from BASE, we use a simple 3-way merge with BASE as the base value.

- 3. Find the unique set of LCA values that do not include the BASE value. If there is only one unique LCA value, we again use three-way merge logic using that unique value as the base.
- 4. At this point, we have determined that we have at least 2 unique values in our LCAs which means that THIS and OTHER would both have to resolve the conflict. If they resolved it in the same way, we would have caught that in step 1. So they either both picked a different LCA value, or one (or both) chose a new value to use.

If OTHER and THIS both picked a different LCA value, we conflict.

If OTHER and THIS both have values that are not LCA values, we also conflict. (Same as 3-way, both sides modified a value in different ways.)

5. (optional) The only tricky part is this: if OTHER has a LCA value, but THIS does not, then we go with THIS, and conversely if THIS has an LCA value, but OTHER does not, then we go with OTHER. The idea is that THIS and OTHER may have resolved things in the same way, and then later changed the value to something newer. (They could have also resolved it differently, and then one side updated again.)

#### InventoryEntry.revision

The last-modified revision for an entry gets treated differently. This is because how it is generated allows us to infer more information. Specifically, any time there is a change to an entry (rename, or content change) the last modified revision is updated. Further, if we are merging, and both sides updated the entry, then we update the last-modified revision at the merge point.

For a picture example:

. A . / \ . B C . \ / . D

For a single entry, the last modified revision in D is:

- 1. A if neither в or c modified it
- 2. B if B modified and c did not
- 3. c if c modified and в did not
- 4. D if B and C modified it

This means that if the last modified revision is the same, there have been no changes in the intermediate time. If OTHER also has the same last modified revision as *any* LCA, then we know that all other LCAs' last-modified revisions are in the ancestry of that value. (Otherwise, when OTHER would need to create a new last modified revision as part of the merge.)

Home | Documentation | Developer Document Catalog (2.2b1) » previous | next
Implementation notes »

In the second se

# Miscellaneous notes

- dirstate An observation re. the dirstate file
- "bzr update" performance analysis "bzr update" performance analysis

♦ Home | Documentation | Developer Document Catalog (2.2b1) »

previous | next

Image: Weight of the second state of the second st

Miscellaneous notes »

# Dirstate

Don't really need the hashes of the current versions - just knowing whether they've changed or not will generally be enough - and just the mtime and ctime of a point in time may be enough?

Home   Documentation   Developer Document Catalog (2.2b1) »	previous   next
Miscellaneous notes »	

 $\bullet$  Home | Documentation | Developer Document Catalog (2.2b1) » Miscellaneous previous notes »

# "bzr update" performance analysis

There are 5 different slightly different situations in which bzr update can be used:

- local only (no-op)
- lightweight checkout
- heavy checkout
- heavy checkout w/ local changes
- bzr update could work on "bound branch" w/no wt

# No new revisions

Should be O(1) to determine Tree base is up to date wt.last-rev == wt.b.last-rev

# No local changes, only new revisions

- 1. Need to move wt.last\_rev (O(1))
- 2. apply delta from base to new rev (O(changes)) applying changes to files is approx (O(lines-in-files ^ 2))
- 3. update meta-info (executable bits, etc) about modified files (O(changes))

2/3 could be concurrent (but that may not necessarily be faster)

potential issue w/ serialized is having 50k files in limbo/

the limbo/ directory could be avoided in some cases, for example when adding new files in new directories.

modifying in place: reduces fragmentation of fs, not atomic w/ local modification, potential of data loss w/o should be safe

"local mod" is diff between disk and last commit, not merge base

Detecting name conflicts should be O(siblings). Alternatively, conflicts with existing files can be detected using stat() and conflicts with new files can be detected by examining the pending transform. This changes complexity to O(changes).

# out of date heavyweight checkout, out of date w/master

- 1. open working tree, check latest revision
- 2. open working tree branch, check latest revision
- 3. mismatch => update wt => wt.b.lastrev apply delta to tree O(changed file size) —- conflicts stop on conflicts stop always -> inform user they need to repeat (why not?, GFD)
- 4. pull new revs M => L O(newrevs)
- apply delta to wt local committed changes become a pending merge local uncommitted stay uncommitted local pending merges are retained (should be gc'd)

offtopic: should bzr update report where the source is ? should bzr update handle both cases (local tree out-of-date w/local branch, checkout out-of-date w/master) ?

if updating would diverge, give opportuniuty to branch/unbind instead local ahead, "push to master"

ideas: 1) can this be done as a single logical step? 2) can this be done w/o modifying working tree until end? possible performance improvements 3) if the pulling revision step could deliver full texts, that may help for the merge (same thing as "bzr pull")

