Bazaar System Administrator's Guide

Release 2.8.0dev1

Bazaar Developers

May 19, 2019

CONTENTS

CHAPTER

ONE

INTRODUCTION

Welcome to the Bazaar Version Control System's guide for system administrators. Bazaar is a flexible system that provides many possible options for serving projects in ways that will hopefully meet your needs. If you have requirements that are not met by the current state of the Bazaar ecosystem, please let us know at bazaar@lists.canonical.com or on Launchpad at https://launchpad.net/bzr.

1.1 Scope of this guide

In this guide, we will discuss various techniques for making Bazaar projects available, migrating from other Version Control Systems, browsing code over the Web and combining Bazaar with other tools. In many of these categories, multiple options exist and we will try to explains the costs and benefits of the various options.

The intended audience for this guide is the individuals who administer the computers that will do the serving. Much of the configuration that we will discuss requires administrator privileges and we will not necessarily indicate every point that those privileges are needed. That said, reading this guide can also be very helpful for those who are interested in communicating to the system administrators about the requirements for making full use of Bazaar.

1.2 What you need to run a Bazaar server

Where possible, we will discuss both Unix (including GNU/Linux) and Windows server environments. For the purposes of this document, we will consider Mac OS X as a type of Unix.

In general, Bazaar requires only Python 2.6 or greater to run. If you would *optionally* like to be able to access branches using SFTP, you need paramiko and pycrypto.

For maximum performance, Bazaar can make use of compiled versions of some critical components of the code. Pure Python alternatives exist for all of these components, but they may be considerably slower. To compile these extensions, you need a C compiler and the relevant header files from the Python package. On GNU/Linux, these may be in a separate package. Other operating systems should have the required headers installed by default.

If you are installing a development version of Bazaar, rather than a released version, you will need Pyrex to create the C extensions. The release tarballs already have the Pyrex-created C files.

CHAPTER

TWO

SIMPLE SETUPS

Consider the following simple scenario where we will be serving Bazaar branches that live on a single server. Those branches are in the subdirectories of `/srv/bzr` (or `C:\bzr`) and they will all be related to a single project called "ProjectX". ProjectX will have a trunk branch and at least one feature branch. As we get further, we will consider other scenarios, but this will be a sufficiently motivating example.

2.1 Smart server

The simplest possible setup for providing outside access to the branches on the server uses Bazaar's built-in smart server tunneled over SSH so that people who can access your server using SSH can have read and write access to branches on the server. This setup uses the authentication mechanisms of SSH including private keys, and the access control mechanisms of the server's operating system. In particular, using groups on the server, it is possible to provide different access privileges to different groups of developers.

2.1.1 Setup

There is no setup required for this on the server, apart from having Bazaar installed and SSH access available to your developers. Using SSH configuration options it is possible to restrict developers from using anything *but* Bazaar on the server via SSH, and to limit what part of the file system they can access.

2.1.2 Client

Clients can access the branches using URLs with the `bzr+ssh://` prefix. For example, to get a local copy of the ProjectX trunk, a developer could do:

```
$ bzr branch bzr+ssh://server.example.com/srv/bzr/projectx/trunk projectx
```

If the developers have write access to the `/srv/bzr/projectx` directory, then they can create new branches themselves using:

```
$ bzr branch bzr+ssh://server.example.com/srv/bzr/projectx/trunk \
bzr+ssh://server.example.com/srv/bzr/projectx/feature-gui
```

Of course, if this isn't desired, then developers should not have write access to the `/srv/bzr/projectx` directory.

2.1.3 Further Configuration

For a project with multiple branches that are all related, it is best to use a shared repository to hold all of the branches. To set this up, do:

```
$ cd /srv/bzr
$ bzr init-repo --no-trees projectx
```

The `--no-trees` option saves space by not creating a copy of the working files on the server's filesystem. Then, any branch created under `/srv/bzr/projectx` (see Migration for some ways to do this) will share storage space, which is particularly helpful for branches that have many revisions in common, such as a project trunk and its feature branches.

If Bazaar is not installed on the user's path or not specified in the SSH configuration, then a path can be specified from the client with the `BZR_REMOTE_PATH` environment variable. For example, if the Bazaar executable is installed in `/usr/local/bzr-2.0/bin/bzr`, then a developer could use:

```
$ BZR_REMOTE_PATH=/usr/local/bzr-2.0/bin/bzr bzr info \
bzr+ssh://server.example.com/srv/bzr/proectx/trunk
```

to get information about the trunk branch. The remote path can also be specified in Bazaar's configuration files for a particular location. See `bzr help configuration` for more details.

If developers have home directories on the server, they can use `/~/` in URLs to refer to their home directory. They can also use `/~username/` to refer to the home directory of user `username`. For example, if there are two developers `alice` and `bob`, then Bob could use:

```
$ bzr log bzr+ssh://server.example.com/~/fix-1023
```

to refer to one of his bug fix branches and:

```
$ bzr log bzr+ssh://server.example.com/~alice/fix-2047
```

to refer to one of Alice's branches. [1]

2.1.4 Using a restricted SSH account to host multiple users and repositories

Once you have a bzr+ssh setup using a shared repository you may want to share that repository among a small set of developers. Using shared SSH access enables you to complete this task without any complicated setup or ongoing management.

To allow multiple users to access Bazaar over ssh we can allow ssh access to a common account that only allows users to run a specific command. Using a single account simplifies deployment as no permissions management issues exist for the filesystem. All users are the same user at the server level. Bazaar labels the commits with each users details so seperate server accounts are not required.

To enable this configuration we update the `~/.ssh/authorized_keys` to include command restrictions for connecting users.

In these examples the user will be called `bzruser`.

The following example shows how a single line is configured:

```
command="bzr serve --inet --allow-writes --directory=/srv/bzr",no-agent-forwarding,no-port-forwardin
```

This command allows the user to access only bzr and disables other SSH use. Write access to each repository in the directory `/srv/bzr` has been granted with `--allow-writes` and can be removed for individual users that should only require read access. The root of the directory structure can be altered for each user to allow them to see only

[1] The version of Bazaar installed on the server must be at least 2.1.0b1 or newer to support `/~/` in bzr+ssh URLs.

a subet of the repositories available. The example below assumes two seperate repositories for Alice and Bob. This method will not allow you to restrict access to part of a repository, you may only restrict access to a single part of the directory structure:

```
command="bzr serve --inet --allow-writes --directory=/srv/bzr/alice/",no-agent-forwarding,no-port-fo
command="bzr serve --inet --allow-writes --directory=/srv/bzr/bob/",no-agent-forwarding,no-port-forwa
command="bzr serve --inet --allow-writes --directory=/srv/bzr/",no-agent-forwarding,no-port-forwardi
```

Alice and Bob have access to their own repository and Repo Manager has access to the each of their repositories. Users are not allowed access to any part of the system except the directory specified. The bzr+ssh urls are simplified by serving using `bzr serve` and the `--directory` option.

If Alice logs in she uses the following command for her fix-1023 branch:

```
$ bzr log bzr+ssh://bzruser@server.example.com/fix-1023
```

If Repo Manager logs in he uses the following command to access Alice's fix-1023:

```
$ bzr log bzr+ssh://bzruser@server.example.com/alice/fix-1023
```

CHAPTER

THREE

OTHER SETUPS

3.1 Dumb servers

Bazaar can also serve branches over protocols that know nothing about Bazaar's specific needs. These are called "dumb servers" to distinguish them from Bazaar's native protocol. Currently HTTP, HTTPS, FTP, SFTP and HTTP+WebDAV can be used to read branches remotely. FTP, SFTP and HTTP+WebDAV can be used for writing as well. To use any of these protocols, it is just necessary to provide access to the server's filesystem under `/srv/bzr`.

For example, for Apache to provide read-only access to the branches in `/srv/bzr` the configuration may look like this:

```
Alias /code /srv/bzr
<Directory /srv/bzr>
    Options Indexes
    # ...
</Directory>
```

and users could use the URL `http://server.example.com/code/projectx/trunk` to refer to the trunk branch.

Note that SFTP access is often available whenever there is SSH access and it may be a good choice when Bazaar cannot be installed on the server to allow `bzr+ssh://` access. Dumb servers are slower by their very nature than the native protocol, but they can be a good choice in situations where the software and protocols that can be used on the server or the network is limited.

3.2 Smart server over HTTP(S)

Bazaar can use its native protocol with HTTP or HTTPS requests. Since HTTP is a network protocol that is available on many networks, this can be a good option where SSH access is not possible. Another benefit of this setup is that all of the authentication and access control methods available to the HTTP server (basic, LDAP, ActiveDirectory, etc.) are then available to control access to Bazaar branches. More information about setting up this type of access using Apache and FastCGI or mod_python or WSGI is in the smart server section of the User's Guide.

3.3 Direct Smart Server Access

The built-in server that is used by `bzr+ssh://` access can also be used as a persistent server on a dedicated port. Bazaar's official port is 4155, although the port used can be configured. Further information on running the Bazaar smart server from inetd, or directly from the shell is in the User's Guide. The dedicated Bazaar server does not currently perform any authentication, so this server by default provides read-only access. It can be run with the

`--allow-writes` option, but the smart server does not do any additional access control so this may allow undesired people to make changes to branches. (Which of course can be reverted.) If the user that runs the server has write access to the branches on the filesystem, then anyone with access to port 4155 on the server can make changes to the branches stored there.

CHAPTER

FOUR

MIGRATION

Migrating between version control systems can be a complicated process, and Bazaar has extensive documentation on the process at http://doc.bazaar.canonical.com/migration/en and we won't attempt to repeat that here. We will try to give a few motivating examples for conversion from Mercurial and Subversion.

4.1 Fast Import

In many projects wishing to use Bazaar, there is pre-existing history for the codebase that should be taken into consideration. Bazaar leverages an interchange format originally developed for Git called *fast-import* to provide migration strategies for many other version control systems. To work with fast-import files, Bazaar needs the fastimport plugin. This can be installed as with any Bazaar plugin.

The way that fast-import can be used for migration is to export the existing history into a fast-import file, then use the `bzr fast-import` command. The *fastimport* plugin includes exporters for Subversion, CVS, Git, Mercurial and darcs, accessible as the `fast-export-from-XXX` commands. Note that `fast-import` should not be used in a branch with existing history.

Assuming that ProjectX was first developed in Mercurial before switching to Bazaar, and that the Mercurial repository is in `/srv/hg/projectx`, the following commands will import that history into a newly created `trunk` branch. (Recall that in Further Configuration we created the `/srv/bzr/projectx` directory as a shared repository.)

```
$ cd /srv/bzr/projectx
$ bzr fast-export-from-hg ../../hg/projectx projectx.fi
$ bzr init trunk
$ bzr fast-import projectx.fi trunk
```

4.2 Subversion Conversion

As the most common centralized version control system, migration from Subversion is particularly important for any *new* version control system. Bazaar's svn plugin provides tools for interaction with Subversion projects. In fact, Bazaar can be used transparently with projects stored in Subversion, but that is beyond the scope of this document. (See http://doc.bazaar.canonical.com/migration/en/foreign/bzr-on-svn-projects.html for more on that subject.) What is relevant here is the `svn-import` command provided by that plugin. This can import an entire subversion repository including tags and branches, particularly if they are stored in Subversion's recommended directory structure: `/tags/`, `/branches/` and `/trunk/`.

This command has flexible ways to specify what paths within the Subversion repository contain branches and which contain tags. For example, the recommended layout for Subversion projects (called `trunk` by the svn plugin) could be specified in `~/.bazaar/subversion.conf` as

```
[203ae883-c723-44c9-aabd-cb56e4f81c9a]
branches = branches/*
tags = tags/*
```

This allows substantially complicated Subversion repositories to be converted into a set of separate Bazaar branches. After installing the svn plugin, see `bzr help svn-import` and `bzr help svn-layout`.

CHAPTER

FIVE

EXTENDING BAZAAR WITH HOOKS AND PLUGINS

Bazaar offers a powerful extension mechanism for adding capabilities. In addition to offering full library API access to all of its structures, which can be useful for outside programs that would like to interact with Bazaar branches, Bazaar can also load *plugins* that perform specific tasks. These specific tasks are specified by *hooks* that run during certain steps of the version control process.

For full documentation on the available hooks, see `bzr help hooks`. Among those, some of the most significant hooks from an administration standpoint are *pre_commit*, *post_commit* and *post_change_branch_tip*. A *pre_commit* hook can inspect a commit before it happens and cancel it if some criteria are not met. This can be useful for enforcing policies about the code, such as line-endings or whitespace conventions. A *post_commit* hook can take actions based on the commit that just happened, such as providing various types of notifications. Finally, a *post_change_branch_tip* hook is a more general form of a *post_commit* hook which is used whenever the tip of a branch changes (which can happen in more ways than just committing). This too can be used for notification purposes, as well as for backups and mirroring.

Information on the whole range of Bazaar plugins is available at http://doc.bazaar.canonical.com/plugins/en/. For purposes of installation, plugins are simply python packages. They can be installed alongside Bazaar in the `bzrlib.plugins` package using each plugin's `setup.py`. They can also be installed in the plugin path which is the user's `~/.bazaar/plugins` directory or can be specified with the `BZR_PLUGIN_PATH` environment variable. See `bzr help configuration` for more on specifying the location of plugins.

5.1 Email Notification

A common need is for every change made on a branch to send an email message to some address, most often a mailing list. These plugins provide that capability in a number of different ways.

The *email* plugin sends email from each individual developer's computer. This can be useful for situations that want to track what each individual developer is working on. On the downside, it requires that every developer's branches be configured individually to use the same plugin.

The next two plugins *hookless-email* and *email-notifier* address this concern by running on a central server whenever changes happen on centrally stored branches.

5.1.1 email

To configure this plugin, simply install the plugin and configure the `post_commit_to` option for each branch. This configuration can be done in the `locations.conf` file or individually in each branch's `branch.conf` file. The sender's email address can be specified as `post_commit_sender` if it is different than the email address reported

by `bzr whoami`. The `post_commit_mailer` option specifies how the mail should be sent. If it isn't set, email is sent via `/usr/bin/mail`. It can also be configured to communicate directly with an SMTP server. For more details on configuring this plugin, see http://doc.bazaar.canonical.com/plugins/en/email-plugin.html. As examples, consider the following two possible configurations. A minimal one (uses `/usr/bin/mail`)

```
[DEFAULT]
post_commit_to = projectx-commits@example.com
```

and a more complicated one (using all of the options)

```
[DEFAULT]
post_commit_url = http://www.example.com/code/projectx/trunk
post_commit_to = projectx-commits@example.com
post_commit_sender = donotreply@example.com
post_commit_mailer = smtplib
smtp_server = mail.example.com:587
smtp_username = bob
# smtp_password = 'not specified, will prompt'
```

5.1.2 hookless-email

This plugin is basically a server-side version of the *email* plugin. It is a program that runs either from the command line or as a daemon that monitors the branches specified on the command line for any changes. When a change occurs to any of the monitored branches, it will send an email to the specified address. Using our simple example, the following command would send an email to `projectx-commits@example.com` on any of the branches under `/srv/bzr` since the last time the command was run. (This command could be set up to run at regular intervals, for example from `cron`.)

```
$ bzr_hookless_email.py --email=projectx-commits@example.com \
--recurse /srv/bzr
```

5.1.3 email-notifier

This is a more elaborate version of the *hookless-email* plugin that can send templated HTML emails, render wiki-style markup in commit messages and update working copies on the server (similar to push_and_update). It can also send emails reporting the creation of new branches or the removal of branches under a specified directory (here `/srv/bzr/projectx`). As it is more complicated, its configuration is also more complicated and we won't repeat its documentation here, but a simple configuration that will send emails on commits and creation/deletion of branches is

```
[smtp]

server=smtp.example.com
# If user is not provided then no authentication will be performed.
user=bob
password=pAssW0rd

[commits]

# The address to send commit emails to.
to=projctx-commits@example.com
from=$revision.committer

# A Cheetah template used to construct the subject of the email message.
subject=$relative_path: $revision_number $summary
```

```
[new-branches]
to=projectx-commits@example.com
from=donotreply@example.com
subject=$relative_path: New branch created

[removed-branches]
to=projectx-commits@example.com
from=donotreply@example.com
subject=$relative_path: Branch removed
```

If this file is stored as `/srv/bzr/email-notifier.conf`, then the command

```
$ bzr-email-notifier.py --config=/srv/bzr/email-notifier.conf /srv/bzr/projectx
```

will watch all branches under the given directory for commits, branch creations and branch deletions.

5.2 Feed Generation

A related concept to sending out emails when branches change is the generation of news feeds from changes on each branch. Interested parties can then choose to follow those news feeds in order to see what is happening on a branch.

5.2.1 branchfeed

This plugin creates an ATOM feed for every branch on every branch change (commit, etc.). It stores these files as `.bzr/branch/branch.atom` inside each branch. Currently, it includes the 20 most recent changes in each feed. To use it, simply install the plugin and set your feed reader to follow the `branch.atom` files.

In addition, there are other tools that are not plugins for creating news feeds from Bazaar branches. See http://wiki.bazaar.canonical.com/FeedGenerators for more on those tools.

5.3 Mirroring

Sometimes it is useful to ensure that one branch exists as an exact copy of another. This can be used to provide simple backup facilities or redundancy (see Back-up and restore for more details on backups). One way to do this using Bazaar's workflows is to make the branch where changes happen into a bound branch of the mirror branch. Then, when commits happen on the working branch, they will also happen on the mirror branch. Note that commits to bound branches do *not* update the mirror branch's working copy, so if the mirror branch is more than just a backup of the complete history of the branch, for example if it is being served as a web page, then additional plugins are necessary.

5.3.1 push_and_update

This plugin updates Bazaar's `push` command to also update the remote working copy. It can only work over connections that imply filesystem or SSH access to the remote working copy (`bzr+ssh://`, `sftp://` and `file://`). Also, it is only useful when the remote branch is updated with an explicit `push` command.

5.3.2 automirror

This plugin is similar to *push_and_update* in that it updates the working copy of a remote branch. The difference is that this plugin is designed to update the remote branch on every change to the working branch. To configure this, set the

`post_commit_mirror = URL` option on a branch. This option can include multiple branch URLs separated by commas to create multiple mirrors. For example, if we want to mirror our `/srv/bzr/projectx/trunk` branch to the URL `sftp://www.example.com/var/www/projectx` (for example if ProjectX were a web project that we wanted to access at `http://www.example.com/projectx`), then we could include

```
[DEFAULT]
post_commit_mirror = sftp://www.example.com/var/www/branches/trunk
```

in the file `/srv/bzr/projectx/trunk/.bzr/branch/branch.conf`.

5.4 Other Useful Plugins

5.4.1 pqm (plugin)

Facilitating interaction with PQM, this plugin provides support for submitting merge requests to a remote Patch Queue Manager. PQM provides a way to automatically run the test suite before merging changes to the trunk branch.

5.4.2 testrunner

Sometimes referred to as the poor man's PQM, this plugin runs a single command on the updated revision (in a temporary directory) and if the command returns 0, then the revision can be committed to that branch. For example, if the testsuite is run with the command `nosetests` in the root of the branch (which returns 0 if the test suite passes and 1 if it doesn't pass), then one can set

```
[DEFAULT]
pre_change_branch_tip_test_command = nosetests
```

in `.bzr/branch/branch.conf`.

5.4.3 checkeol

This plugin is an example of a *pre_commit* hook that checks the revision being committed for meeting some policy. In this case, it checks that all of the files have the specified line endings. It uses a configuration file `.bzreol` in the root of the working tree (similar to the `.bzrignore` file). This configuration file has sections for line feed endings (LF), carriage return/line-feed endings (CRLF) and carriage return endings (CR). For an unusual example that specifies different line endings for different files, that file might look like

```
[LF]
*.py
*.[ch]

[CRLF]
*.txt
*.ini

[CR]
foo.mac
```

or if you simply want to enforce a single line ending convention on the branch you can use

```
[LF]
*
```

This plugin needs to be installed on the server where the branch updates will happen, and the `.bzreol` file must be in each branch where line ending policies will be enforced. (Adding it to the branch with `bzr add .bzreol` is an easy way to ensure this, although it means that branches on the server must have working trees.)

5.4.4 text_checker

This plugin is a more advanced version of *checkeol* that can check such coding style guidelines such as trailing whitespace, long lines and files that don't end with a newline. It is configured using Bazaar's built in rules specification in `BZR_HOME/rules` (see `bzr help rules` for more information. For different types of undesired changes, you can specify different types of actions. For example

```
[name NEWS README]
trailing_whitespace=fail
long_lines=warn
newline_at_eof=ignore

[name *.py]
tabs=fail
long_line_length=78
long_lines=fail
trailing_whitespace=fail
```

will prevent changes from adding new trailing whitespace to the specified files and keep all python source files free of tabs and lines over 78 characters. To commit while violating these rules, one can pass the `--text-check-warn-only` option to commit.

CHAPTER

SIX

WEB-BASED CODE BROWSING

Browsing the history of a project online is an important part of version control, since it allows people to easily see what happens in a branch without having to have a local, up-to-date copy of that branch. There are a number of possible choices for browsing Bazaar branches on the web, but we will cover one of them in particular detail and briefly mention the other choices where they differ.

6.1 Loggerhead

Loggerhead is a code browsing interface for Bazaar branches (now used in Launchpad). To see an example of Loggerhead in action, browse to http://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev/files which is the loggerhead view of Bazaar's trunk branch. Loggerhead runs as a web application on the server which is accessed over HTTP via a RESTful interface. It is possible to run this application on its own dedicated port as `http://www.example.com:8080` or to proxy this location behind a separate web server, for example at `http://www.example.com/loggerhead/`. We will discuss both of these configurations below.

6.1.1 Requirements

Loggerhead depends on a number of other Python packages for the various Web technologies that it builds on. Some of these must be installed to use loggerhead, although some of them are optional. From the loggerhead *README* file, these are

1. SimpleTAL for templating. On Ubuntu, *sudo apt-get install python-simpletal* or download from http://www.owlfish.com/software/simpleTAL/download.html
2. simplejson for producing JSON data. On Ubuntu, *sudo apt-get install python-simplejson* or use *easy_install simplejson*.
3. Paste for the server. (You need version 1.2 or newer of Paste.) On Ubuntu, *sudo apt-get install python-paste* or use *easy_install Paste*
4. Paste Deploy (optional, needed when proxying through Apache) On Ubuntu, *sudo apt-get install python-pastedeploy* or use *easy_install PasteDeploy*
5. flup (optional, needed to use FastCGI, SCGI or AJP) On Ubuntu, *sudo apt-get install python-flup* or use *easy_install flup*

Although directions for installing these on Ubuntu are given, most other GNU/Linux distributions should package these dependencies, making installation easy. For Windows and Mac OS X, they should all be `easy_install`-able or at worst installable from the Python sources.

6.1.2 Built-in Web Server

Loggerhead has a built-in web server and when started with the `serve-branches` command, that web server is started on a default port listening on the localhost. If port 8080 (the default) is accessible on `www.example.com`, then running

```
$ serve-branches --host=www.example.com --port=8080 /srv/bzr
```

will list all of the available branches under that directory on `http://www.example.com:8080/`, so that the ProjectX trunk could be browsed at `http://www.example.com:8080/projectx/trunk`. Note that loggerhead provides HTTP access to the underlying Bazaar branches (similar to that described in Smart server over HTTP(S)), so this command should be run as a user without write privileges in `/srv/bzr`. By default, loggerhead only listens on the localhost, not any external ports, unless specified as above.

6.1.3 Behind a Proxy

A more common and more safe way to run loggerhead is behind another web server which will proxy certain requests to the loggerhead server on the localhost. To do this, you need to have PasteDeploy installed (see Requirements). Assuming that your server has Apache running, you need to add configuration such as this to set up the proxy

```
<Location "/loggerhead/">
    ProxyPass http://127.0.0.1:8080/
    ProxyPassReverse http://127.0.0.1:8080/
</Location>
```

If your proxy runs at some path within the server, then the `serve-branches` command must be started with the `--prefix` option. For this example, we could start loggerhead with the command

```
$ serve-branches --prefix=/loggerhead /srv/bzr
```

This would allow the trunk branch of ProjectX to be browsed at `http://www.example.com/loggerhead/projectx/trunk`.

Loggerhead comes with a script allowing it to run as a service on `init.d` based Unix systems. Contributions to do a similar thing on Windows servers would be welcomed at http://launchpad.net/loggerhead.

6.2 Other web interfaces

There are a number of other web interfaces available for Bazaar branches (see the list at http://wiki.bazaar.canonical.com/WebInterfaces) and we will just mention a couple of them here for their advantages in particular situations.

trac+bzr (http://launchpad.net/trac-bzr) Trac is a popular web app that integrates a browser for branches, an issue tracker and a wiki. trac+bzr is a trac extension that allows for the trac to be used with Bazaar.

webbzr (http://thoughts.enseed.com/webbzr) This is a notable solution because it is written in pure PHP for web hosts that don't provide a way to run arbitrary Python applications such as Trac or Loggerhead.

Redmine (http://redmine.org/) Like trac, Redmine is a full project management application using the Ruby on Rails framework. It includes support for Bazaar branches.

CHAPTER

SEVEN

INTEGRATION WITH OTHER TOOLS

CHAPTER

EIGHT

SECURITY

8.1 Authentication

Bazaar's philosophy on authentication is that it is best to reuse existing authentication technologies, rather than trying to reinvent potentially complicated methods for securely identifying users. As such, we describe two such uses of existing software for authentication purposes.

8.1.1 Using SSH

SSH is a very well tested and featureful technology for authenticating users. For situations where all of the developers have local accounts on the server, it is trivial to provide secure, authenticated `bzr+ssh://` access. One concern with this method is that it may not be desirable to grant shell access to developers on the server machine. In this case, Bazaar provides `bzr_ssh_path_limiter`, a script that runs the Bazaar smart server on the server machine at a specified path, and allows no other access.

To set it up, specify:

```
command="/path/to/bzr_ssh_path_limiter <path>" <typical key line>
```

in each user's `~/.ssh/authorized_keys` file on the server, where *<path>* is the path to limit access to (and its subdirectories). For more documentation on the syntax of the `authorized_keys` file see the documentation of the SSH server. This will only permit Bazaar access to the specified path and no other SSH access for that user.

If it isn't desired to give each user an account on the server, multiple private/public key pairs can be included under one single SSH account (say sshuser) in the `~sshuser/.ssh/authorized_keys` file and then each developer can be given their own private key. They can then use `bzr+ssh://sshuser@server.example.com/` URLs to access the server.

8.1.2 Using HTTP authentication methods

8.2 Access Control

Many projects need fine-grained access control on who may read and write to which branches. Incorporating these controls into OS-level user accounts using groups and filesystem permissions can be difficult or even not permitted in some instances. Bazaar provides a script called `bzr_access` that can be used to provide access control based on usernames, with authentication performed by SSH. To do so, we need to set up private-key authentication in SSH. This can be done using a single SSH user on the server, or one account per user. The idea is to use the SSH's `authorized_keys` file to specify the `bzr_access` script as the only command that can be run by a user identified by a particular key pair.

First, you will need to generate a private/public key pair for each user who will be accessing the repository. The private key should be distributed to the user and the public key will be needed on the server to identify the user. On the server, in the SSH user's `~/.ssh/authorized_keys` file, use the following line for each repository user and the corresponding public key:

```
command="/path/to/bzr_access /path/to/bzr /path/to/repository <username>",no- port-forwarding,no-X11-
```

where <*key*> is the (possibly very long) public key, <*type*> is the type of SSH key and <*username*> is the username to associate with that public key.

The `bzr_access` script obtains its configuration information from the file `/path/to/repository/bzr_access.conf`. This file should not be placed under version control in a branch located at `/path/to/repository` since that would allow anyone with access to the repository to change the access control rules. The `bzr_access.conf` file is in a simple INI-style format with sections defined by `[groups]` and `[/]`. The options in the `[groups]` section are the names of groups and the values of those options should be the usernames in that group. Inside the `[/]` section, the options are usernames or group names (prefixed with `@`) and the values are either `rw`, `r` or nothing, representing read-write access, read-only access or no access at all. A sample of `bzr_access.conf` could be:

```
[groups]
admins = alpha
devels = beta, gamma, delta

[/]
@admins = rw
@devels = r
upsilon =
```

where the user whose key is associated with *alpha* would have read-write access, the users *beta*, *gamma* and *delta* would have read-only access and user *upsilon* would not be able to access any branches under `/path/to/repository`.

8.2.1 Additional Considerations with `bzr_access`

As currently written, `bzr_access` only allows each public key to be associated with a single repository location. This means that if developers need to access two or more different repositories, then each developer will need to have two or more private keys for SSH and be able to select between them (see `man ssh` for more information on configuring multiple private keys).

Also, each repository can only have a single configuration file, with access configured for all branches in the repository. This means that if different access rules are needed for different projects, then those projects must be in different repositories. This then necessitates the use of multiple private keys as just described.

Finally, as noted above under Using SSH all of the public keys may be included in the `authorized_keys` file of a single user on the server. It is also possible to use a single private/public key pair for all of the developers, but this only allows a single username for access control to the repository (since the username is associated with the public key in `authorized_keys`. While this is certainly possible it seems to defeat the purpose of fine-grained access control, although it does provide the same limited SSH access as that described above.

CHAPTER

NINE

BACK-UP AND RESTORE

Backing up Bazaar branches can be done in two different ways. If an existing filesystem-based backup scheme already exists, then it can easily be used where the Bazaar branches reside. Alternately, Bazaar itself can be used to mirror the desired branches to or from another location for backup purposes.

9.1 Filesystem Backups

Bazaar transactions are atomic in the sense that the disk format is such that it is in a valid state at any instant in time. However, for a backup process that takes a finite amount of time to complete, it is possible to have inconsistencies between different on-disk structures when backing up a live branch or repository. (Bazaar itself manages this concurrency issue by only *reading* those structures in a well-defined order.) Tools such as LVM that allow instantaneous snapshots of the contents of a disk can be used to take filesystem backups of live Bazaar branches and repositories.

For other backup methods, it is necessary to take the branch or repository offline while the backup is being done in order to guarantee consistency between the various files that comprise a Bazaar branch's history. This requirement can be alleviated by using Bazaar as its own backup client, since it follows an order for reading that is designed to manage concurrent access (see the next section for details). Depending on the different access methods that are being used for a branch, there are different ways to take the branch "offline". For `bzr+ssh://` access, it is possible to temporarily change the filesystem permissions to prevent write access from any users. For `http://` access, changing permissions, shutting down the HTTP server or switching the server to a separate configuration that disallows access are all possible ways to take a branch offline for backup. Finally, for direct filesystem access, it is necessary to make the branch directories un-writable.

Because this sort of downtime can be very disruptive, we strongly encourage using Bazaar itself as a backup client, where branches are copied and updated using Bazaar directly.

9.2 Bazaar as its own backup

The features that make Bazaar a good distributed version control system also make it a good choice for backing itself up. In particular, complete and consistent copies of any branch can easily be obtained with the `branch` and `pull` commands. As a result, a backup process can simply run `bzr pull` on a copy of the main branch to fully update that copy. If this backup process runs periodically, then the backups will be as current as the last time that `pull` was run. (This is in addition to the fact that revisions are immutable in Bazaar so that a prior revision of a branch is always recoverable from that branch when the revision id is known.)

As an example, consider a separate backup server that stores backups in `/var/backup`. On that server, we could initially run

```
$ cd /var/backup
$ bzr branch bzr+ssh://server.example.com/srv/bzr/trunk
$ bzr branch bzr+ssh://server.example.com/srv/bzr/feature-gui
```

to create the branches on the backup server. Then, we could regularly (for example from `cron`) do

```
$ cd /var/backup/trunk
$ bzr pull  # the location to pull from is remembered
$ cd ../var/backup/feature-gui
$ bzr pull  # again, the parent location is remembered
```

The action of pulling from the parent for all branches in some directory is common enough that there is a plugin to do it. The bzrtools plugin contains a `multi-pull` command that does a `pull` in all branches under a specified directory.

With this plugin installed, the above periodically run commands would be

```
$ cd /var/backup
$ bzr multi-pull
```

Note that `multi-pull` does a pull in *every* branch in the specified directory (the current directory by default) and care should be taken that this is the desired effect. A simple script could also substitute for the multi-pull command while also offering greater flexibility.

9.2.1 Bound Branch Backups

When `bzr pull` is run regularly to keep a backup copy up to date, then it is possible that there are new revisions in the original branch that have not yet been pulled into the backup branch. To alleviate this problem, we can set the branches up so that new revisions are *pushed* to the backup rather than periodically pulling. One way to do this is using Bazaar's concept of bound branches, where a commit in one branch happens only when the same commit succeeds in the branch to which it is *bound*. As a push-type technology, it is set up on the server itself rather than on the backup machine. For each branch that should be backed up, you just need to use the `bind` command to set the URL for the backup branch. In our example, we first need to create the branches on the backup server (we'll use `bzr push`, but we could as easily have used `bzr branch` from the backup server)

```
$ cd /srv/bzr/projectx/trunk
$ bzr push bzr+ssh://backup.example.com/var/backup/trunk
$ cd ../feature-gui
$ bzr push bzr+ssh://backup.example.com/var/backup/feature-gui
```

and then we need to bind the main branches to their backups

```
$ cd ../trunk
$ bzr bind bzr+ssh://backup.example.com/var/backup/trunk
$ cd ../feature-gui
$ bzr bind bzr+ssh://backup.example.com/var/backup/feature-gui
```

A branch can only be bound to a single location, so multiple backups cannot be created using this method.

Using the automirror plugin mentioned under Hooks and Plugins, one can also make a push-type backup system that more naturally handles mutliple backups. Simply set the `post_commit_mirror` option to multiple URLs separated by commas. In order to backup to the backup server and a remote location, one could do

```
$ cd /srv/bzr/trunk
$ echo "post_commit_mirror=bzr+ssh://backup.example.com/var/backup/trunk,\
bzr+ssh://offsite.example.org/projectx-corp/backup/trunk" >> .bzr/branch/branch.conf
$ cd ../feature-gui
```

```
$ echo "post_commit_mirror=bzr+ssh://backup.example.com/var/backup/feature-gui,\
bzr+ssh://offsite.example.org/projectx-corp/backup/feature-gui" >> .bzr/branch/branch.conf
```

As for any push-type backup strategy that maintains consistency, the downside of this method is that all of the backup commits must succeed before the initial commit can succeed. If there is a many mirror branches or if the bound branch has a slow network connection, then the delay in the original commit may be unacceptably long. In this case, pull-type backups, or a mixed system may be preferable.

9.3 Restoring from Backups

9.3.1 Checking backup consistency

Many a system administrator has been bitten by having a backup process, but when it came time to restore from backups, finding out that the backups themselves were flawed. As such, it is important to check the quality of the backups periodically. In Bazaar, there are two ways to do this: using the `bzr check` command and by simply making a new branch from the backup. The `bzr check` command goes through all of the revisions in a branch and checks them for validity according to Bazaar's internal invariants. Since it goes through every revision, it can be quite slow for large branches. The other way to ensure that the backups can be restored from is to perform a test restoration. This means performing the restoration process in a temporary directory. After the restoration process, `bzr check` may again be relevant for testing the validity of the restored branches. The following two sections present two restoration recipes.

9.3.2 Restoring Filesystem Backups

There are many different backup tools with different ways of accessing the backup data, so we can't cover them all here. What we will say is that restoring the contents of the `/srv/bzr` directory completely will restore all branches stored there to their state at the time of the backup (see Filesystem Backups for concerns on backing up live branches.) For example, if the backups were mounted at `/mnt/backup/bzr` then we could restore using simply:

```
$ cd /srv
$ mv bzr bzr.old
$ cp -r /mnt/backup/bzr bzr
```

Of course, to restore only a single branch from backup, it is sufficient to copy only that branch. Until the restored backup has been successfully used in practice, we recommend keeping the original directory available.

9.3.3 Restoring Bazaar-based Backups

In order to restore from backup branches, we can simply branch them into the appropriate location:

```
$ cd /srv
$ mv bzr bzr.old
$ cd bzr
$ bzr branch bzr+ssh://backup.example.com/var/backup/trunk
$ bzr branch bzr+ssh://backup.example.com/var/backup/feature-gui
```

If there are multiple backups, then change the URL above to restore from the other backups.

CHAPTER

TEN

UPGRADES

Bazaar has a strong commitment to inter-version compatibility both on disk and over the network. Newer clients should be able to interact with older versions on the server (although perhaps not at optimal speed) and older clients should also be able to communicate with newer versions of Bazaar on the server. Divergences from this rule are considered bugs and are fixed in later versions.

That said, Bazaar is constantly improving and the most recent versions are the most featureful and have better performance. In particular, the Bazaar versions 2.0 and later have significant advantages over earlier versions, including a more compact disk format, faster network operations and overall performance improvements. With the 2.0 release, Bazaar has moved to a stable/development release model where the 2.x series is maintained with bugfixes releases for six months, while simultaneously the 2.(x+1) series is being developed with monthly beta releases that are suitable for everyday use. Bazaar development has a stable trunk with an extensive test suite, so there is no reason to fear using the development series for everyday use, it simply changes more often than the stable series. Many users do run the development version of Bazaar and update it regularly, including most of the Bazaar developers themselves.

10.1 Software upgrades

Upgrading the Bazaar software is as simple as re-installing the Python package using either the latest binary package for Windows or Mac OS X, the binary package provided by your GNU/Linux distribution, or installing from the source release. See http://wiki.bazaar.canonical.com/Downloads for the latest releases for all supported platforms.

Bazaar's later versions support all of the earlier disk formats (back to the very first one), so there is no need to upgrade the branches on the disk when upgrading the software. To make use of particular new features that might need updated versions on both the server and developer's machines, it does not matter if the clients or the servers are upgraded first.

10.2 Disk format upgrades

In its evolution, Bazaar has used a sequence of disk formats for improved storage efficiency and speed. With the new disk format released in version 2.0, there is a commitment to keep that disk format until version 3.0 is released, which has not even been planned yet. (Bazaar 2.0 was released almost two years after Bazaar 1.0.) As a result, disk format upgrades should be extremely infrequent.

If there are existing branches in an older format that you would like to upgrade to the latest format, you can see the 2.0 Upgrade Guide for more information. From the system administration perspective, it is important to coordinate the timing of various upgrades in the process. First, the central branches on the server should be upgraded. Next, any local mirrors that developers have should be upgraded. Finally, developers' local branches should be upgraded. These upgrades will require an appropriate version of the software whenever they are performed. (It is possible to upgrade branches remotely over the network, but it may be much slower.)

10.3 Plugin upgrades

When Bazaar does update its version, plugins that use the Bazaar API may need to be upgraded to reflect changes in that API. Some plugins have strict version dependencies on the version of the Bazaar API that they will accept. If this is the case, then you should ensure that the plugins you depend on have been updated *before* you upgrade your Bazaar version to avoid a situation where your plugins won't work with the installed version of Bazaar. If this does happen, then the solution is simply to reinstall the previous version of Bazaar that *did* work with the plugins. For installations that depend on a large number of plugins, this sort of version upgrade should be tested in a safe sandbox to ensure that the entire collection of Bazaar and its plugins can all work together.

CHAPTER

ELEVEN

ADVANCED TOPICS

11.1 System Monitoring

11.2 Capacity Planning Tips

11.3 Clustering

11.4 Multi-site Setups

The "distributed" in distributed version control system should indicate that Bazaar is well suited for multi-site development situations and indeed, that is the case. The advantage comes from the ease and transparency of managing merges between branches with divergent history. Note that there are many, many different ways to manage widely-flung development setups using Bazaar and its branching and merging capabilities. These can be discovered and tested before being implemented as policy. We will describe one such possible setup here.

Consider ProjectX Corp's international expansion with a new branch office in Darwin, Australia, in addition to the company's headquarters in Austin, Texas, USA. One of the difficulties of a far-flung multi-site development environment such as this is that the network connection between Australia and Texas is slow and unreliable. So, each branch office would like the master branch to be local to them. (In situations with good network connectivity, a local branch bound to the remote master may be all that is needed to support multi-site development.)

Of course, with two master branches, there is always the question of which one is authoritative. Given Bazaar's facility at managing multiple branches, we suggest that it is best not to privilege either the Texas or Australia branches, but to merge both of them into a separate master branch (which may reside at either site). For definiteness, we will locate the master branch at the Texas site. So, we will have three branches stored on two servers: trunk and texas-integration at the Texas site and australia-integration at the Darwin site. These branches are named in terms of the sites where the development takes place, but in many cases it may make more sense to name branches after the functional teams rather their geographical locations. Since we are trying illustrate the issues with multi-*site* development, we will persist in this naming scheme.

11.4.1 Setup

Using our previous setup at the Texas site, we will simply rename the old trunk branch as trunk and branch a copy as texas-integration.

```
$ cd /srv/bzr/projectx
$ mv trunk trunk              # can simply rename on the filesystem
$ bzr branch trunk texas-integration   # very fast in a shared repository
```

In Australia, we need to set up the `/srv/bzr/projectx` directory and get a copy of the current trunk as australia-integration:

```
$ mkdir -p /srv/bzr
$ cd /srv/bzr
$ bzr init-repo --no-trees projectx
$ cd projectx
$ bzr branch bzr+ssh://server.example.com/srv/bzr/trunk
$ bzr branch trunk australia-integration
```

11.4.2 Merging to master

Then, each office works with their local copy of the trunk. At some point, sooner or later depending on the pace of development in the two locations, the two local trunks need to be merged. (In general, sooner beats later when merging, since there is no penalty for multiple merges.) In this example, Alice at the Texas office will do the merging on her local machine using branches on the server:

```
# Get a copy of the Australia branch in Texas.  After the initial branch
# command, use pull to keep the branch up to date.  With a slow network,
# this is the only slow part
$ bzr branch bzr+ssh://autralia.example.com/srv/bzr/projectx/australia-integration \
  bzr+ssh://server.example.com/srv/bzr/projectx/australia-integration

# Check out the master branch locally for doing the merge
$ cd ~/projectx
$ bzr checkout bzr+ssh://server.example.com/srv/bzr/projectx/trunk
$ cd trunk
$ bzr merge bzr+ssh://server.example.com/srv/bzr/projectx/texas-integration
# Run the test suite and resolve any conflicts
$ bzr commit -m "Merge Texas branch to master"

# Now, merge from Australia using the local copy of that branch
$ bzr merge bzr+ssh://server.example.com/srv/bzr/projectx/australia-integration
# Run the test suite and resolve any conflicts between the two offices
$ bzr commit -m "Merge Australia branch to master"
```

Note that Bazaar does not commit even cleanly applied merges by default. This is because although a merge may apply cleanly, the merged state still needs to be checked before it is committed. (Just because there are no text conflicts does not mean that everything will work after a merge.) An alternative that can pull when possible and merge otherwise is available with `bzr merge --pull`.

11.4.3 Merging back to local trunks

Now the trunk branch is the most up-to-date version of the software and both of the local trunks need to reincorporate the changes from the master. If no new commits have been made to texas-integration, then that can happen using `bzr pull`:

```
$ cd ~/projectx
$ bzr checkout bzr+ssh://server.example.com/srv/bzr/projectx/texas-integration
$ cd texas-integration
$ bzr pull ../trunk  # Use trunk from the local disk
                           # No need to commit
```

If new changes have happened on texas-integration since the integration with trunk, then the above pull will produce an error suggesting to use merge:

```
$ bzr merge ../trunk
# Run test suite, resolve conflicts
$ bzr commit -m "Merging Australian changes"
```

In Australia, they will need to update their local copy of trunk:

```
$ cd /srv/bzr/projectx/trunk
$ bzr pull     # parent location is used by default
```

Then, they need to pull or merge the changes from trunk into the local trunk. This should be done by a developer with a checkout of australia-integration so that they can run the test suite:

```
$ cd ~/projectx
$ bzr co bzr+ssh://australia.example.com/srv/bzr/projectx/australia-integration
$ cd australia-integration
$ bzr merge bzr+ssh://australia.example.com/srv/bzr/projectx/trunk
# Run test suite and integrate Texan changes with only recent local
# development
$ bzr commit -m "Integrate work from Texas"
```

11.4.4 Other Considerations

Multi-site deployments can be complicated, due to the many possible variations of development velocity, divisions of labor, network connectivity, resources for integration, etc. The preceding description is meant to be one possible way to do fairly symmetric multi-site development. (Neither Texas or Australia is privileged in this structure.) In a situation where there is one main site and other smaller sites, one of the local trunk branches can be eliminated and trunk can be used directly for development at the main site.

It is also up to the particular situation how frequently the local trunks are integrated into the master trunk. Given resources specifically for integration, it is conceivable that a developer may be constantly responsible for integrating changes from the two teams. Alternatively, the two sites could work on well-separated, well-defined features and merge to the master trunk only when their respective features are complete. Given the difficulty of resolving conflicts in very large merges and the ease of merge handling in Bazaar, we suggest that merges be done more frequently, rather than less.

CHAPTER

TWELVE

LICENCE

Copyright 2008-2011 Canonical Ltd. Bazaar is free software, and you may use, modify and redistribute both Bazaar and this document under the terms of the GNU General Public License version 2 or later. See <http://www.gnu.org/licenses/>.