Source Code Management at HET

Introduction

What This Document is For

The intent of this document is to describe the current standards and recommendations for source code management at HET, and to provide a basic guide for the use of version control repository software. The topic of software deployment (aka. “installation”) is related, and will be covered in its own section.

This document is a work in progress, and is intended to be updated as changes are adopted in our recommended practices. It is recommended to avoid making or distributing copies of this document; instead keep a link to it, to ensure you’re reading the current version.

Intended Audience

This document is intended for employees developing software, libraries, scripts, and other functional routines for use at the Hobby Eberly Telescope. Very little assumption should be made of existing software development knowledge, as the audience is not restricted to professional software programmers. As a result, some material in this document may be excessively elementary or simplistic for some readers.

Document History

This document was originally written in October 2019 as a summary for night staff, to introduce version control concepts and practices, and their application in the HET environment.

Some Definitions

Several terms are used in this document which may be new, or may benefit from clarification due to ambiguous typical use.

Version, revision: A “version” or “revision”, in the context of “version control”, is any change made to source and *committed* to a version control repository. When the commit process is completed, a new revision number is generated to identify the changes that occurred with that commit. Confusingly, a separate concept of a “version” also exists with respect to the software development process, the versions of released software (1.0, 2.58.15b, “2019” etc) with which users may be immediately familiar. Version control systems may offer some incidental features or

best practices to manage these release versions but this is a different matter, largely unrelated to the “version” concept in the phrase “version control”. The word “revision” is sometimes used instead to describe these incremental repository updates, to reduce this ambiguity.

Repository: A repository is a managed database containing the full history of edits to committed files. A version control system manages repositories such that files can be retrieved from them at any time, in the condition in which they existed at the time of any commit since the repository was created. A repository can (and usually does) contain multiple simultaneous parallel versions of the source files, for example if edits are being made simultaneously by multiple developers. A *repository server* is a computer system on which the version control software has been installed, which hosts one or more repositories. This server may be a physical computer or a VM.

Code: Source files containing procedures which are translated by a compiler, interpreter or shell into machine instructions for a computer to execute, or such content collectively contained within these files. Traditionally in the computer science and software development jargon, “code” has been an uncountable/substance noun. Recently, “codes” has emerged as a commonly encountered plural, especially among non-professional developers. It may be gaining some popularity with professionals as well.

What Is Version Control?

Version control, also known as “revision control”, is a technique of storing changes to sets of files that are edited repeatedly and progressively over time, and is typically applied to “source code” in a software development operation. Superficially this serves purposes similar to making regular backup copies of files, but results in much greater control over the editing process, more sophisticated opportunities to recover from unwanted changes, and better organization.

What is Source?

The concept of “source code” or simply “source(s)” and the distinction from “binaries”, “executables” or (more generally) “deliverables” is immediately intuitive to those working in compiled languages such as C++. For those who only work with interpreted scripts (eg. bash, Python, Perl, TCL), there may be little or no physical difference between source that is edited and the files that are installed. Nonetheless, a difference should be observed in practice -- scripts should not be edited “in place” (in the directory where they are intended to be installed in production) but rather in a separate development location, where they can be tested and validated before deployment.

For the purposes of this document, “source” is the collection of files directly edited by human activity. This is typically in the form of ASCII or Unicode text files, containing any of the following content:

- Computer instructions to be compiled into binary executable code

- Computer instructions to be executed by an interpreter
- Markup or specially annotated written text to be converted into formatted documents
- Graphical, audio or other media resources to be included in the compilation of a computer application or documentation
- Data files edited by humans (directly or via a software tool), intended for the consumption of an application or script, and deployed with the software

Importantly, source **should NOT** include the following:

- Data or object files generated by the application or compilation process
- Files that are wholly derivative of human-edited files. For example, editing a GUI resource in a graphical editor may create or modify a file containing the graphical design, and also trigger a step that generates C++ code. In this case, the graphical editor's native GUI resource file would be considered "source", but the automatically generated C++ code would not. The latter is *code,* but it is not *source code.*
- Binary object files, executables, libraries, formatted documents or other files produced by a compiler or other source processor.

Working on source

In order for the practices of using version control to make sense, we must first recognize some standards in software development, which are assumptions under which version control tools like Subversion and Git are designed.

Source is a collection of files. Because source commonly comprises multiple related files and other resources, these files are typically contained in a single directory, which is itself often organized with multiple subordinate directories. The phrase "source tree" refers to this organization, and the outermost directory that contains it.

Source is separate. It may be very convenient to edit a script in the location where it will be executed, and run it from this location during development in order to test it. This technique, apart from being largely incompatible with version control, creates problems for keeping the process of updating working production code clear and organized. The same logic applies to code in a compiled language, which a developer might expect to build such that the compiled binaries end up in the location from which they will be executed as a direct output of the build script. This is less likely to happen in practice, but is best avoided for the same reasons.

Source should contain the means for its compilation and deployment. Since we must accept that source will be edited (and possibly compiled) in a location separate from where it is executed in normal practice, there must be a well-defined process for transferring scripts or compiled programs into their correct target location. The version control system doesn't know or care what you use. Often this will be some form of *makefile*, a text description for the `make` utility to compile and install your program. Other more elaborate frameworks exist, such as GNU Autotools or CMake. At the very least, this recommendation could be met by a document describing to a human how to build and install the software, and in this case should assume a minimum of contextual knowledge on the part of those who might later need to follow it. In any case, these files should be considered part of the source.

Source should be protected from change. It may seem strange that we must protect source code from the very means by which it is created and evolved -- changes made by development activity. What this really means is that your edits must be revocable, in some cases long after they're made. The simplest way to achieve this is by making repeated backup copies of your code as you work, but this creates opportunities for confusion and disorganization, and is not readily applicable to collaborative development, or backing out changes earlier than the latest update.

Source for a project must have one authoritative location and revision history. This can sometimes be a hassle to keep straight when making simple backup copies of your source tree whenever changes are made, and can utterly fall apart when collaborating on a project with other developers, without some intelligent means of coordination.

How Can Version Control Help?

A version control system solves the problems of managing change over time, and protecting the integrity and history of edits to sources.

Protection from changes

When your source tree is protected by a version control system, the files you add and the changes you make are committed to a *repository*. As long as this repository exists, the change history is never lost. Every commit to the repository is recorded permanently in the form of a "change set", containing only the changes made since the previous commit. In this way, the result is effectively similar to a backup copy made every time changes are committed, but with far greater storage efficiency and much more sophisticated control over how these changes can be later backed out and undone if needed.

Complex version history — branching and merging

Version control can provide great control over versions along a single timeline of changes; changes can be deleted if they're later found to cause problems, and old removed changes can be reapplied.

But a major source of power and flexibility in modern version control systems is the ability to split the development timeline into *branches*, and intelligently *merge* the changes made in these branches back into the version that is to be delivered. This provides flexibility for collaborative development by allowing developers to work in their own branches, and allows an individual developer or a team to make experimental changes that can be easily discarded if they're not successful.

Version Control at HET

Subversion

The current version control software used at HET is called “Subversion”, often referred to as the name of its client executable on Unix systems, `svn`. On Windows, the client is called TortoiseSVN. The user interface of TortoiseSVN is graphical but largely analogous to that of `svn`.

What appears to be a minor detail in our usage of Subversion actually has a substantial effect on how the software works. At HET, we exclusively use the “svn+ssh” protocol, which carries the Subversion client’s communications with the repository server over a Secure Shell or “ssh” encrypted connection. The result is that the client software is doing most of the work, and repository database access and user identity management is handled at the filesystem level on the server computer.

The Repository Server

At HET our primary repository server is called **`ute2`**, and this name will appear in the URL that is created for your software repositories. There is another repository server you may encounter called **`hetdex-pr`** which currently contains TCS among other things, though generally there should be no new repositories created on this server.

Login access

To begin with, all Subversion users will need to be able to log in to ute2 via ssh:

```
$ ssh username@ute2.as.utexas.edu
```
or
```
$ ssh ute2
```

As in the examples, the username and full domain name of the server are optional. Include the username if your account username on ute2 differs from that on your local workstation.

If you are unable to log in, contact an administrator (currently Stephen Cook or Chris Robison) to get access.

SSH keys

Using SSH for access to the Subversion repository server provides many security and administrative benefits, though it does introduce complications as well. When operating via SSH, Subversion can sometimes perform the login sequence multiple times for a single user task, each requiring an authentication step. When using passwords for authentication, this can quickly become quite tedious and irritating, especially with graphical tools like Tortoise which may need to

log in many times in order to update their user interface. The best means to streamline the use of Subversion with SSH is through the use of “keys” which enable logins without passwords.

Keys are generated in pairs, with one key being public and the other private. By placing your public key in your user account on the server, you allow the SSH server to authenticate with your private key on your workstation, without a need for a password at each login attempt. The basic process to generate and distribute keys is described in Appendix A of this document.

A strong password is highly recommended to protect your private key! When the private key is used to connect to a server (such as ute2 in this case), by default the password will be requested to unlock the key, and it seems that this would defeat the purpose of using keys as a convenience to log in. However, SSH provides a system for handling this, called an *agent*. Using an agent, you can unlock and load a private key into memory once, and then use it from that point on until you log out of your local session. There are many different forms of SSH-compatible agents, some are included with desktop environments like Gnome, some are options for password management systems like KeePass, and the basic SSH software (OpenSSH) on Unix operating systems has an agent (`ssh-agent`) as well.

On modern systems, it is highly likely that you have an agent already loaded. If you try to list the loaded keys:

```
$ ssh-add -l
```

and you get an error connecting to the agent (“*Could not open a connection to your authentication agent*”), then there is likely no agent running. At a minimum, you can execute

```
$ ssh-agent bash
```

to run it, though it may be worth troubleshooting as the lack of a running agent by default on a modern Unix or Linux OS may indicate other problems. Any other message, including “*The agent has no identities*” indicates that the agent is running in the background. Simply execute `ssh-add` with no parameters to load and unlock your primary key, assuming it’s been stored using a standard filename like `id_rsa`.

On Windows, the most commonly used software for SSH is called PuTTY, and its agent is called Pageant. Tortoise is able to use keys stored with Pageant, as well as host aliases in PuTTY to access repositories.

Creating a Repository

For the Git version control system there are multiple web-based self-service tools for repository management like GitHub, GitLab, BitBucket and so forth. Unfortunately, recent searches for similar tools supporting Subversion have not yielded any viable candidates. Therefore, managing repositories on ute2 requires administrative access to the server and so for the moment the result is that repositories for HET software will need to be created by request of an administrator — Stephen Cook or Chris Robison.

Repository scope and definition

Prior to making your request for a repository, consider overall what the repository will be for. There are few rules for making this decision, but there are some guidelines worth considering that will influence whether some sources should be combined, or kept separate.

- A repository should typically contain the sources for a single *deliverable*. This is the software that should be installed at one time, for a cohesive set of purposes. Think of this deliverable like a shrink-wrapped software product, with its own release history, public version sequence, and development timeline.
- There are cases where it may make sense to package multiple closely-related deliverables in a single repository. For example a software application may make use of a library of core features, but the library may have a documented interface (API) and may change much less frequently than the code in the application, so it is practical to manage their deployments separately. If the application is the sole user of the library, then putting them in the same repository may be reasonable. In this case, divide the repository into two separate trees at the outermost directory. In this way, the two deliverables can be easily moved into separate repositories later (for example, if other applications are developed that use the library).
- It may be convenient to combine similar small programs into a single repository, especially if they have some common usage context. This can include programs that perform similar tasks, or which are used in a common job role or operation. While it's not inconceivable to create a repository to manage a single script for example, such situations should compel some thought about how functional commonality may be found with other such scripts or programs. Be aware that management in a single repository would imply a common version history across all its contents, and either a single or separated deliverables as described above.
- There are cases where separating deliverables and repositories makes sense. In cases where software components are developed with formalized and documented interfaces and their dependencies form clear hierarchies, better organization can be achieved by keeping them in separate repositories. Examples include libraries of code shared by multiple separate applications or programs, and server processes and daemons that communicate with other programs through a documented network protocol. Look for opportunities to reduce the rebuild and re-delivery of software that hasn't actually changed. This may sometimes require multiple builds and installations from multiple repositories for a given software update, but in practice this encourages more careful consideration about the design of shared code and modular software.

The repository tree

It may seem intuitive that a repository should contain a tree of files and directories identical to the tree of source contained within it. And in practice it does, but this tree of source code isn't normally kept directly at the base of the repository's tree. Instead, common Subversion convention

involves three base directories, containing many apparent copies of the source tree: `trunk`, `branches`, and `tags`.

`trunk/` Within this directory is your set of files and subdirectories you would expect to see in your source tree. There are multiple conventional interpretations of the purpose and usage of trunk; two common approaches could be called “latest code” (code is merged here continually, and this contains the latest developments that at least correctly build) and “latest release” (trunk contains a copy of the most recent released version).

`branches/nnn/` Each directory under branches contains a version of the code created to develop a feature or capability or to correct a problem, a summary of which should be expressed in the directory name. Be generously descriptive here. `branches/change_to_tabbed_ui` is a clearer summary to other developers than `branches/gui_work`. Branches can be copies of trunk, copies of other branches, and copies of tags (for maintenance fixes of older releases). The structure under `branches` can also be further subdivided; for example, a developer on a team project could create a directory for all his/her branches, or group multiple branches together, that all apply to a major feature change. As the work performed in a branch is completed, the changes in the branch are merged to trunk, to the source branch, or to a new tag. The branch is then deleted.

`tags/nnn/` When a new release is deployed, the version being released is copied to tags. Each directory here should be named according to the release numbering scheme chosen for the repository. This can involve formal major and minor version numbers, but for smaller projects will often just be a date. In this case it is recommended to use descending order (YMD) dates, eg. a release made on January 17 2020 might be stored in `tags/20200117/` or similar. The directories under tags are generally not deleted, so this list continues to grow over time.

Initial checkout: Create your working copy

Once your repository has been created, you will be given a URL to access it. This will look something like

```
svn+ssh://ute2/MyProject
```

This URL will point to a real directory on the repository server, but one that should not be accessed directly, as it contains a database accessible only to the Subversion software. Instead, you’ll need to use Subversion’s checkout, update, commit and other operations to interact with the repository. This starts with the “checkout”, which makes a special directory on your local workstation. This “working copy” directory will contain your code, but also additional hidden files managed by Subversion which describe its state and relationship to the repository server. To check out your new repository, go into a directory you’ve set up to work on all your projects, and execute something like the following example:

```
$ svn co svn+ssh://ute2/MyProject
```

If successful, svn will reply with “`Checked out revision 0`” and create a directory called MyProject on your local computer within the current directory from which the command was

executed. Importantly, that directory will contain hidden data linking it with the repository, such that many future operations won't need to specify its URL.

In the above command, the command "`co`" is an abbreviation for `checkout` (either can be used). This command is a good example of normal svn syntax— "`svn`", followed by a command, followed by that command's parameters.

We've just checked out a newly created, empty repository with no contents. In the usual case when checking out an existing repository, the resulting working copy directory would be filled with files representing the entire development history of the project (and so for large projects, checking out the entire repository can be risky, as the resulting working copy could quickly fill up your local disk).

However, since we've just checked out a new repository, the `MyProject` directory will contain nothing but Subversion's hidden files. It's effectively empty, at "revision 0", ready to create or import your project.

If you wish to create your local working copy directory with a name different from the name in the repository URL, you can provide it on the command line:

```
$ svn co svn+ssh://ute2/MyProject my_project
```

The following examples will assume the use of the same name as on the repository server.

Repository Operations

In what follows, we'll be working with an example series of commands that highlights some of the most-used functionality for basic source code management, scratching the surface of what the Subversion command line client can do. This is intended as a place to start, but you'll inevitably need to look up more details as you use it.

Getting help

You can request a list of commands that the Subversion client supports:

```
$ svn help
```

And then you can get a reference to the options and functionality of any particular command, such as `status`:

```
$ svn help status
```

For a more technically in-depth but highly readable tour through Subversion's features, the canonical guide is *Version Control with Subversion*, aka. "The SVN Book", which is available in print or for free online:

http://svnbook.red-bean.com/

Minimal tree setup

To begin with, we should at the very minimum create a trunk directory in our working copy, and create or import source files into it.

```
$ cd MyProject
$ mkdir trunk
$ touch trunk/my_test_script
```

These commands create the trunk directory within MyProject and one empty example file within it, which for the moment represents the contents of your source tree.

Status

```
$ svn status
?      trunk
```

The `status` command is one of the most frequently used, as it tells you the state of your working copy, in particular what has changed since the last update from the repository. Two things are notable in the response from the status command. First, the `trunk` directory is reported, with a question mark preceding it. This means the trunk directory has been found in the working copy, but not yet added to version control. The second thing to note is that the file `my_test_script` wasn't reported. The status command doesn't enter into directories that are not in version control.

This is an important behavior to be aware of. Often while writing and building software, compilers and other development tools will create files within the source tree that should not be included in version control, and you may create them too, for temporary use, or perhaps as inputs or outputs of your software as you test it. To help avoid including anything in the repository that doesn't need to be there, Subversion does not include files by default just because they're in the working copy. It needs to be told which files to manage explicitly. This only needs to be done once for each file and/or directory.

Later on, you'll find other characters preceding files and directories in the output of the status command. Some common ones you'll see most frequently:

- A File to be added to the repository on next commit
- D File to be deleted from the repository on next commit
- M File has been modified; modifications will be sent on next commit
- I File is being ignored due to repository configuration
- ! File under version control has been deleted locally and is missing
- C File has a conflict from the last update

Adding files to version control

```
$ svn add trunk
```

```
A           trunk
A           trunk/my_test_script
```

The add command instructs Subversion to place the selected files and directories under version control. When a directory is specified, it is implied to add all the files it contains.

Running the status command again, we get the same output. In both cases, the “A” indicates that the working copy contains files that will be added to the repository at the next commit.

```
$ svn status
A     trunk
A     trunk/my_test_script
```

Committing changes to the repository

Everything we’ve done so far since the initial checkout has been within our working copy. The status command checks the state of the working copy, the add command tells the local Subversion client about files it needs to track, and so these amount to local configuration and content operations. To preserve your changes and make all of this worth it, we need to commit these changes to the repository.

```
$ svn commit
```

By default, the above command will not work! You’ll get the following output:

```
svn: E205007: Commit failed (details follow):
svn: E205007: Could not use external editor to fetch log message; consider
setting the $SVN_EDITOR environment variable or using the --message (-m) or
--file (-F) options
svn: E205007: The EDITOR, SVN_EDITOR or VISUAL environment variable or
'editor-cmd' run-time configuration option is empty or consists solely of
whitespace. Expected a shell command.
```

Subversion intentionally makes it difficult to make commits without some kind of comment, and it is highly recommended to always provide one. This comment serves to explain and summarize the intent of the editing activity being committed, and is extremely useful in making sense of a project’s development history months or years later. You have three options to perform commits properly.

```
$ svn commit -m “Initial commit, created trunk and first test file”
```

You can include the comment explicitly on the command line surrounded by double quotes, which is the simplest (but not necessarily the easiest or best) option. This tends to limit the length of comments in practice, and makes more descriptive multiline comments impractical.

```
$ svn commit --editor-cmd vim
```

This command allows the Subversion client to launch an external editor, allowing you to compose a commit message in a more convenient tool of your choice. At the end of the file in the editor, It will add a list of files included in the commit, following a separator line beginning with two dash characters.

```
Initial commit, created trunk and first test file.

--This line, and those below, will be ignored--

A     trunk
A     trunk/my_test_script
```

Type your message into the top of the file presented by the editor. By default, the separator line and the file list will not be included in the commit message. It is customary at HET to **delete** the separator line before committing, which causes Subversion to include the file list below it in the message, which will then be visible in the repository logs.

The more descriptive you are with your message, the more helpful the commit logs will be later when a developer (probably someone else) is looking through them.

Save the file and exit the editor to continue the commit.

```
$ svn commit
```

There is a way to make this simple version of the command work, and that is by setting the `SVN_EDITOR` environment variable as suggested in the above error message. This is the recommended approach, and makes the commit process relatively easy and seamless. Once this variable is set, ideally in an initialization script so it is always available, Subversion will launch the selected editor by default when using the commit command unless it's overridden with the --editor-cmd option as above. For example, you could add the following line to your `~/.bashrc` file:

```
export SVN_EDITOR=/usr/bin/vim
```

GUI editors can be used instead, as long as you'll be working in an environment where a graphical (X) display is available:

```
export SVN_EDITOR=/usr/bin/gedit
```

Another advantage of using an editor, either specified on the command line or via the SVN_EDITOR variable, is that you are given a chance to abort if you have second thoughts while reviewing the file list or editing your message. Simply exit the editor without saving the file, and Subversion will notice and ask if you wish to abort, continue committing, or go back to editing the message.

After the commit completes, you'll have something like the following output:

```
Adding      trunk
Adding      trunk/my_test_script
Transmitting file data .
Committed revision 1.
```

At this point, the `status` command will return with no output; there are no outstanding files according to the local Subversion client.

Getting working copy information

At this point, we'll interrupt the flow of the discussion with a small tangent.

```
$ svn info
Path: .
Working Copy Root Path: /home/chris/dvl/src/MyProject
URL: svn+ssh://ute2/repos/MyProject
Repository Root: svn+ssh://ute2/repos/MyProject
Repository UUID: c788832c-8638-4136-8c2b-87ea99d8a1ac
Revision: 0
Node Kind: directory
Schedule: normal
Last Changed Rev: 0
Last Changed Date: 2019-11-11 17:14:37 -0500 (Mon, 11 Nov 2019)
```

Notice the revision number is still at zero! After a commit, the repository is updated and a new revision number is generated, but some information in the commit doesn't officially make it back to the client to be stored in the local working copy. Despite the message at the end of the commit, this includes the new revision number, and as a result it also affects the commit logs (covered later).

Generally, make note of the info command as it is very handy for reporting exactly how your working copy is currently configured. This will become more useful when we begin redirecting or "switching" our working copy to other places in the repository.

Updating

The update command pulls any changes or differences in the repository to your working copy, so that your working copy matches. This can include differences in file content, but also can include file and directory properties, merge info, and other metadata.

```
$ svn update
Updating '.':
At revision 1.
```

Now if you try the `info` command again, it will report the correct revision number. There's more going on with this command however, which is important if you are collaborating with others, or if you're working from multiple different computers (eg. your office workstation and your home PC or laptop). The `update` command will fetch all changes from the repository which do not exist in your local working copy, bringing the two fully in sync with each other.

Branching

When working on a project, you may want to use Subversion to help separate an editing activity into a new space, especially if that activity is experimental, and/or must proceed while the current, unchanged software must be maintained for bug fixes, etc if necessary in the meantime.

We'll follow the common Subversion convention, and create a branches directory. This would typically be created at the same time as trunk, immediately after creating the repository. We'll do it now:

```
$ mkdir branches tags
$ svn add branches tags
Svn commit -m "Adding branches and tags directories"
Adding      branches
Adding      tags

Committed revision 2.
```

Now we can create a branch, in which we'll do further development on our example. This involves making a copy of the existing source tree, but **not locally**. Instead we tell Subversion to make the copy on the server. The result is that the copy is created, but initially consumes almost no extra space on the server. Subversion realizes there is no difference — they're still exact copies, and so it just makes note of that, but presents the result as if a normal copy has happened.

$ svn copy ^/trunk ^/branches/adding_new_file

There's a lot to unpack here; let's go through it.

- The parameters to the copy command as we've used it here are actually URLs, but abbreviated with Subversion's caret (^) syntax. It's optional but makes typing this sort of command a lot more convenient. The caret expands to the URL for the base directory of the repository as pointed to by the working copy. So `^/trunk` is interpreted as `svn+ssh://ute2/repos/MyProject/trunk`, in this case.
- By giving a URL for the source and destination, we're telling Subversion to do a server-side copy.
- Because this operation happens in the repository directly (instead of needing to be committed later), it has the same effect as a commit. In particular, you'll be prompted to edit a commit message, or will need to specify one on the command line with -m if you haven't defined SVN_EDITOR.

- As with the commit above, the change has happened in the repository and isn't yet reflected in your working copy. Use an update command to sync them.

If you take a look at your entire working copy at this point, you'll notice you now have both copies of your source stored locally:

```
$ tree
.
├── branches
│   └── adding_new_file
│       └── my_test_script
├── tags
└── trunk
        └── my_test_script
```

This illustrates an important point. While Subversion has the ability to make *cheap copies* that store only differences and changes within its database on the server, your local filesystem has no such sophistication. The working copy will contain a fully-expanded copy of your repository as of the current revision.

Looking around in the repository

Let's take a look at the branch directory we've just created in the repository on the server.

```
$ svn ls ^/branches
adding_new_file/
```

The `ls` command is analogous to the `ls` command in your local shell; it lists files in the repository at the given URL.

```
$ svn ls ^/
branches/
tags/
trunk/
```

Note that when specifying the root of the repository, the trailing slash is needed. Otherwise Subversion will not expand the caret into a URL, and will interpret the caret as a literal local filename in the working copy.

Switching your working copy

In our example so far, our project's entire source tree is represented by a single file that we've called my_test_script, and that file is currently empty. This is nearly the smallest imaginable project, but of course real projects won't be nearly so small. It may quickly become cumbersome to have an entire project's repository, and all its mostly redundant branched and tagged copies on your workstation, and there's usually no need for that. Fortunately Subversion provides a tool that allows you to focus your working copy on just the branch you're actually working on.

First, we'll try switching to focus on trunk exclusively.

```
$ svn switch --ignore-ancestry ^/trunk
D     trunk
D     branches
D     tags
A     my_test_script
Updated to revision 3.
```

Subversion shows the local changes it needed to make in order to convert your working copy to reflect only the contents of trunk. If you now look in your working copy, you will see only your source code (my_test_script), rather than the lowest-level directories.

Prior to Subversion v7, the `switch` command worked with any change between directories. This meant you could switch between dissimilar directories with the same syntax as switching between copied branches. In versions after 7, you must specify the `--ignore-ancestry` parameter to perform switches between directories that are not "ancestrally related" -- directories that did not originate as copies of each other at some point in the past. This applies to situations like the above, switching from a full copy of the repository to a specific branch or trunk.

While we're here, let's make a trivial edit to the file, adding a *shebang* (`#!/bin/bash`) as the first line.

```
$ vim my_test_script
$ svn commit -m "Modifying my_test_script, added shebang"
Sending     my_test_script
Transmitting file data .
Committed revision 4.
```

Having made a change in trunk (let's imagine this is a quick bug fix to current code), let's do some work on our feature branch.

```
$ svn switch ^/branches/adding_new_file
U     my_test_script
Updated to revision 4.
```

Notice that the `--ignore-ancestry` parameter was not needed this time; this will be the typical case when using `switch` to change between different branches and `trunk`. Notice also that `my_test_script` is again empty (0 bytes). It was updated during the switch (the "U" above) to the version that existed in `trunk` at the time this branch was copied from `trunk`; that will come up again later. For now, let's do the work intended for this branch.

```
$ vim another_test_script
$ svn add another_test_script
A           another_test_script
$ svn commit -m "Added another file."
Adding      another_test_script
```

```
Transmitting file data .
Committed revision 5.
```

At this point you have made two changes to your source, on two parallel timelines of development. One in `trunk` (editing `my_test_script`) and the other in a branch (creating `another_test_script`). The edit in trunk may have been done prior to the edit in the branch, but effectively their order is ambiguous and will likely in reality consist of multiple edits and commits, interleaved in time. Subversion keeps track and allows you to bring the two variants together.

Merging

Merging is among the most powerful features offered by modern version control software. It can be a very complex process, but Subversion attempts to hide most of the details and figure things out intelligently. There is some work that the user needs to do, but this can be fairly easily summarized for most cases.

In Subversion, there are two kinds of merges. The first is called a *sync merge*, and this is done to keep your branch in sync with changes that have happened in its source. In a collaborative environment, this will happen as other developers modify trunk or the branch from which your development branch was created. Those changes need to be merged into your working copy from time to time to make sure you're not making incompatible changes, and in any case, even when working alone, it will need to be done one last time when the work in your branch is complete.

A sync merge from trunk to the current branch (`adding_new_file`) is performed as follows:

```
$ svn update
Updating '.':
At revision 5.
$ svn merge ^/trunk
--- Merging r3 through r5 into '.':
U    my_test_script
--- Recording mergeinfo for merge of r3 through r5 into '.':
 U   .
$ svn commit -m "Merging changes from trunk"
Sending     .
Sending     my_test_script
Transmitting file data .
Committed revision 6.
```

Again a lot going on here.

- It's always a good idea to perform an update immediately prior to a merge. Subversion will refuse to perform the merge if the working copy isn't fully updated.
- The merge contains the changes that were made to the originating branch (`trunk` in this case) *after* this branch was created or since the last merge. Here that includes the edit to `my_test_script` that was performed in `trunk`.

- The result of the merge command is effectively an edit to the files and directories in your working copy -- this operation does not write to the repository automatically. So you can verify not only that the merge completed, but that the resulting code builds properly. This means a commit is required to save the merged code in the repository, where it will become part of the change revision history of the branch.

The sync merge process described above may be performed multiple times during the lifetime of a branch, to keep it up-to-date with respect to whatever it was copied from, but eventually you will need to merge the contents of the branch back to its originating branch or to trunk. This may be the kind of operation, called a reintegration, that may primarily come to mind when you think of “merging” in general.

It’s important to note that a reintegration merge must start with a sync merge as described above, and it is in the sync merge that any conflicts or other problems must be resolved. Once the sync merge has been performed successfully, the reintegration can proceed. First, we must switch to trunk:

```
$ svn update
Updating '.':
At revision 6.
$ svn switch ^/trunk
D     another_test_script
 U   .
Updated to revision 6.
```

Notice that in order to convert the working copy from the `adding_new_file` branch to `trunk`, the file `another_test_script` created in the branch is deleted, but since the change in `my_test_script` was merged to the branch, it’s the same and is therefore untouched during the switch. Now we can perform the merge from the branch.

```
$ svn merge --reintegrate ^/branches/adding_new_file
--- Merging differences between repository URLs into '.':
A     another_test_script
--- Recording mergeinfo for merge between repository URLs into '.':
 U   .
$ svn commit -m "Merged adding_new_file branch to trunk"
Sending     .
Adding      another_test_script

Committed revision 7.
$ svn update
Updating '.':
At revision 7.
```

In Subversion clients from version 1.8, the `--reintegrate` parameter is no longer necessary; assuming v1.7 and earlier, it advises the client on how to record the merge history (“*mergeinfo*”) in the repository, and is needed to distinguish the reintegration from a sync merge.

Now that the merge is complete, you should be able to verify that all the changes made in both trunk and branches/adding_new_file have been incorporated. The branch could be used for continued development, but in most cases it will no longer be useful and should be discarded.

Deleting branches

To delete the feature branch after it is no longer being used, issue the following:

```
$ svn rm ^/branches/adding_new_file

Committed revision 8.
$ svn update
Updating '.':
At revision 8.
```

Providing the rm command a URL recursively removes the directory and its contents at that URL. This action is performed on the repository server, so once again a comment is needed, either in an editor or with a -m parameter on the command line. An update is also needed after the deletion.

Viewing revision history

Now that we have created some history with file additions, edits, branching and merging, let's look at the logs created by these activities.

```
$ svn log
------------------------------------------------------------------------
r7 | chris | 2019-11-13 11:27:09 -0500 (Wed, 13 Nov 2019) | 1 line

Merged adding_new_file branch to trunk
------------------------------------------------------------------------
r4 | chris | 2019-11-12 17:18:37 -0500 (Tue, 12 Nov 2019) | 1 line

Modifying my_test_script, added shebang
------------------------------------------------------------------------
r1 | chris | 2019-11-12 10:49:22 -0500 (Tue, 12 Nov 2019) | 6 lines

Initial commit, created trunk and first test file


A     trunk
A     trunk/my_test_script

------------------------------------------------------------------------
```

By default, the log command runs as if its parameter were the current directory. In this case it lists the history of changes that affect the current directory. To limit the listing to a maximum number of messages, you can use the -l parameter.

```
$ svn log -l 2
------------------------------------------------------------------------
r7 | chris | 2019-11-13 11:27:09 -0500 (Wed, 13 Nov 2019) | 1 line

Merged adding_new_file branch to trunk
------------------------------------------------------------------------
r4 | chris | 2019-11-12 17:18:37 -0500 (Tue, 12 Nov 2019) | 1 line

Modifying my_test_script, added shebang
------------------------------------------------------------------------
```

To see the history for a particular file or directory, you can provide it to the command.

```
$ svn log another_test_script
------------------------------------------------------------------------
r7 | chris | 2019-11-13 11:27:09 -0500 (Wed, 13 Nov 2019) | 1 line

Merged adding_new_file branch to trunk
------------------------------------------------------------------------
r5 | chris | 2019-11-12 17:37:12 -0500 (Tue, 12 Nov 2019) | 1 line

Added another file.
------------------------------------------------------------------------
```

Notice that this includes commit messages not displayed in the log listing for the parent directory.

Local file operations

At times you may need to move or delete files in your source tree which are under version control. Though you can do this directly on the filesystem via the normal commands and then inform Subversion of the changes afterward, it can be simpler and less confusing to perform these operations through Subversion in one step.

```
$ svn mkdir scripts
A           scripts
$ ls
another_test_script  my_test_script  scripts
$ svn mv my_test_script scripts
A           scripts/my_test_script
D           my_test_script
$ svn commit -m "Moved my_test_script to new scripts directory"
Deleting    my_test_script
```

```
Adding      scripts
Adding      scripts/my_test_script

Committed revision 9.
$ svn update
Updating '.':
At revision 9.
```

The `mkdir` and `mv` commands are again analogous to their corresponding shell commands, while updating Subversion's hidden metadata in the working copy.

```
$ svn rm another_test_script
D           another_test_script
$ svn commit -m "Removing another_test_script"
Deleting    another_test_script

Committed revision 10.
$ svn update
Updating '.':
At revision 10.
```

The `rm` command deletes the file from the source tree. Remember that files deleted from Subversion are still retained in the repository in earlier revisions. Files cannot be truly, physically deleted from a repository after they're committed. This is why it is important to avoid accidentally committing files -- especially very large ones -- that don't belong. Committed files will remain in the repository for as long as it exists.

Changing to earlier revisions

To prove the above, you can change to an earlier revision using the update command, giving that revision as a parameter.

```
$ svn update -r 9
Updating '.':
A     another_test_script
Updated to revision 9.
$ ls
another_test_script  scripts
$ svn update -r 8
Updating '.':
D     scripts
A     my_test_script
Updated to revision 8.
$ ls
another_test_script  my_test_script
$ svn update
Updating '.':
D     another_test_script
```

```
D     my_test_script
A     scripts
A     scripts/my_test_script
Updated to revision 10.
```

Like the switch command, update makes the necessary changes, additions and deletions to convert the working copy to match what is in the repository at the requested revision.

This won't be needed often in practice, but seeing it in action reinforces our understanding of how the repository software works and what it is capable of.

Viewing local changes

Sometimes it is helpful to review what you have changed, line-by-line, in your source files before you commit them. Let's make a change, adding a line of text to a file and then having Subversion show us the difference.

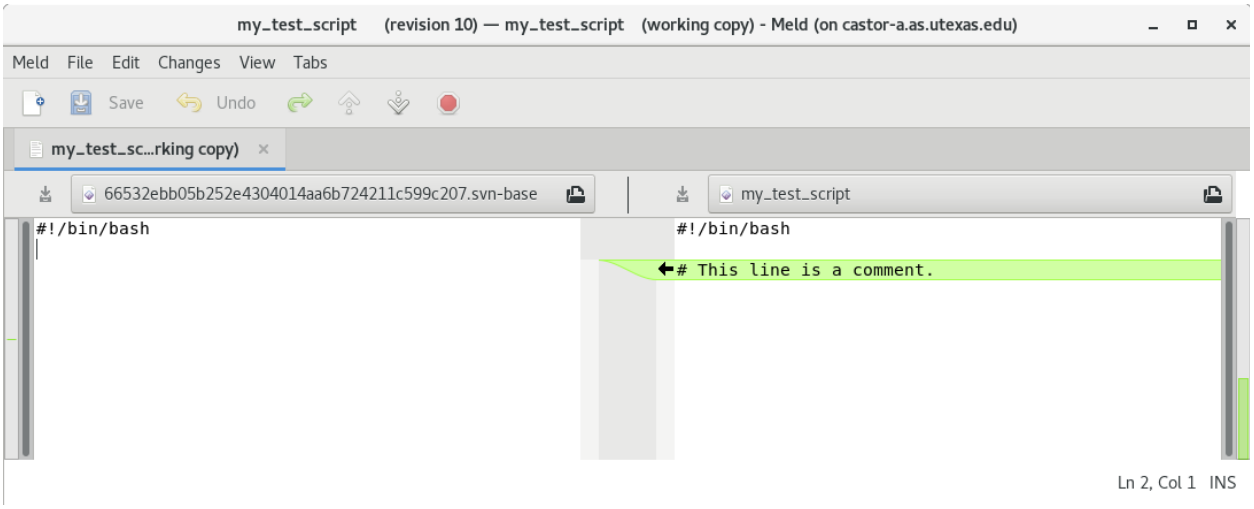
```
$ vim scripts/my_test_script
$ svn status
M     scripts/my_test_script
$ svn diff scripts/my_test_script
Index: scripts/my_test_script
===================================================================
--- scripts/my_test_script    (revision 10)
+++ scripts/my_test_script    (working copy)
@@ -1,2 +1,3 @@
 #!/bin/bash

+# This line is a comment.
```

The above output shows that a line (`This line is a comment.`) has been added to my_test_script.

The annotated text is in what is called "unified diff format", similar to that produced by executing `diff -u` on the command line. For long files with multiple complex edits, this format will become frustrating to interpret for those lacking extensive familiarity with the format. A graphical tool like Meld is highly recommended, to make the differences easier to see. (Meld can be installed by the software package manager on most Linux-based operating systems, eg. `yum`, `dnf` or `apt`.)

```
$ svn diff --diff-cmd meld scripts/my_test_script
```

When telling Subversion to use an external diff application, it launches that application and supplies it with the files to compare as parameters. One of these will be a "pristine" copy of the file from the repository, contained in Subversion's hidden directory.



The graphical application can then show the difference in a visually intuitive way. Once we're satisfied with the correctness of the change, we can commit it.

Releasing (tagging) your project

Subversion and version control software in general is mostly unconcerned with the actual method used to deploy software, but there is a convention for managing these releases in the repository's software development timeline. Assuming that the latest code that's being prepared for release is in `trunk`, the release process should involve making a permanent labeled copy of that code to a special place in the tree. This label is called a *tag*.

```
$ svn cp ^/trunk ^/tags/20191113 -m "Tagging 2019-11-13 release"


Committed revision 12.
```

This is the same operation as what is used to create a branch. By putting the copy in the `tags` directory, we're signifying to other developers that this particular copy of `trunk` is important, and will not be edited or removed as branches are. We label the tag with the version number, in this case using the date of the release.

Location, Integrity and Duplication of Source Files

Developing source managed by a version control system is in some ways not much different from choosing not to use one -- you're still working in a local directory, on local files. The development process itself is largely unchanged. You write your code, test and debug it, prepare your installation and deploy the software. What is affected most by the use of version control is the processes and risks in the context of that development.

When you're developing without version control, *the files you're editing* are important. They must be protected from loss, and the effort to do so is very likely to be exclusively your own. The disk on your laptop or workstation may fail; mistakes may introduce unintended changes, development efforts may turn out to be fruitless or wrongheaded and may need to be abandoned.

The responsibility for recovering from these situations is especially important to keep in mind when developing on your own equipment.

Without version control, the *location of the files* is also important, especially if you would like to develop your software on multiple platforms, switching perhaps between a laptop, home PC, and an office workstation. If you have your software in multiple locations, you must keep track of which is the latest or authoritative code, and copy or synchronize between machines as needed.

Under version control, these concerns don't go away, but they are managed to an extent that by following good practices, they become much less important. The integrity of the working copy is far less important -- you could lose your working copy, and the entire development PC at any point, and your development efforts are secure, up to the last commit.

The location is unimportant, as long as you can connect to the repository. You can check out your working copy to any directory you want, on any computer, at any time. You can commit and delete your working copy, and recreate it with another checkout. You can have working copies in multiple locations simultaneously, and synchronize through commits and updates.

Remote Access to the Repository Server

When writing software at HET, using a computer with a wired connection to the network, access to our "ute2" repository server is straightforward. Outside of the network, eg. from home, it's a little more complicated.

What is required in all cases is a connection to the SSH (Secure Shell) daemon on ute2. This is carried over a TCP port, number 22 (the standard port number for SSH communication). When you are not on the HET network, you will not be able to connect to this computer directly; you will not see its IP address. You will need to connect to an intermediate computer, one that is located on the internal network but also has an externally visible IP address. You'll then need to "tunnel" through that connection to reach ute2.

What follows will discuss these steps using OpenSSH software (`ssh`) on Unix/Linux operating systems. On Windows, the most common SSH software is called PuTTY, and its configuration options are similar.

Connecting to the gateway

The local gateway at HET is a computer known internally as ursa.as.utexas.edu, at 192.168.66.2. This name is applied to the internal network interface and its private address. Externally, the computer has a public IP address (206.76.137.130), which resolves from the public hostname het.as.utexas.edu. Despite the two names for the same computer, we'll refer to it hereafter as "ursa".

From the public internet, you can establish a secure connection to this computer:

```
$ ssh username@het.as.utexas.edu
```

This gets you into ursa, and from there you can use its internal network connection to connect to any internal machine including ute2. But this by itself isn't helpful if you want to develop on your

own computer. Somehow we need to make ute2 accessible directly. To do this, we can add a “tunnel” via the -L (local forward) option.

```
$ ssh -L 10022:ute2:22 username@het.as.utexas.edu
```

This command line connects to ursa, and then forwards the port 10022 on your local PC, through ursa, to port 22 on the host ***it*** sees as “ute2”. So a connection to your own port 10022 will behave like a connection to ute2, port 22. The port number 10022 is chosen mostly randomly, though it’s a good idea to choose non-system port numbers (greater than 1024) unless you have specific reasons to do otherwise.

Connecting through the tunnel

Once the above connection is established, you can minimize or set that terminal aside, and open another. In the new local terminal, you can connect to ute2 directly:

```
$ ssh username@localhost -p 10022
```

In the above, “username” should be the user name recognized by ute2. As usual, it can be omitted if it is the same as your local username.

Once you have demonstrated that you can connect to ute2 through the tunnel, you can use this to check out from a repository server. Using our example repository from earlier, we modify the URL to use our tunnel:

```
$ svn co svn+ssh://username@localhost:10022/repos/MyProject
```

And once the checkout completes successfully, that URL is remembered in the working copy metadata, so it won’t need to be specified again. We’ll have the prerequisite of opening our tunnel connection to ursa first, but from that point Subversion operations will work as we expect.

Saving the tunnel configuration

You may want to save the tunnel configuration to make it more convenient to start the tunnel in the future. You can do this by setting up an entry in the OpenSSH client’s configuration file.

```
$ vim ~/.ssh/config
```

In place of `vim`, use whatever editor you like. In this configuration file, you can save the essential options and provide an alias that will refer to them on the ssh command line.

```
Host ursa
      HostName het.as.utexas.edu
      User username
      Port 22
      LocalForward 10022 ute2:22
```

Now to create the tunnel, you can just connect to your new alias:

```
$ ssh ursa
```

The tunnel will be created according to the options in the config file, and you can minimize the terminal and use Subversion as above.

The laptop problem

A problem is encountered in the situation that you have a single portable computer which sometimes has a wired ethernet connection to the internal network, and also sometimes connects from home or elsewhere, and then lacks a direct connection to ute2. The problem is that your working copy is created with the URL you specify, and that URL is retained within the working copy's hidden metadata. The technique described above involves using a different URL depending on whether your access to ute2 is from the internal network, or from the outside, thus making it invalid when changing from internal to external network connections or vice versa. While Subversion provides a means to change the URL for a working copy via its `relocate` command, this method is cumbersome and not without some risk of error.

We can resolve the problem by creating connection-specific versions of the SSH configuration file, taking advantage of SSH aliases, which happen to work for Subversion just as they work for SSH.

Consider two SSH configuration files. First, we create `~/.ssh/config_hetlocal`

```
Host ursa
      HostName ursa.as.utexas.edu
      User username
      Port 22

Host ute2
      HostName ute2.as.utexas.edu
      User username
      Port 22
```

Then we can create another file, `~/.ssh/config_hettunnel`

```
Host ursa
      HostName het.as.utexas.edu
      User username
      Port 22
      LocalForward 10022 ute2:22

Host ute2
      HostName localhost
      User username
      Port 10022
```

Having both of these files, we can alternately overwrite `~/.ssh/config` with one or the other and by doing so, we select what connection context we have -- internal to HET, or outside the network. For example, let's "activate" our tunnel configuration for outside work. Assuming we're connected from outside the office:

```
$ cp ~/.ssh/config_hettunnel ~/.ssh/config
$ ssh ursa
```

This will set up tunneling for external use of the ute2 repository. Notice what this latter configuration file does. It associates the alias `ute2` with our local tunnel connection. This means that once the tunnel is set up by connecting to ursa (the second command above), we can use `ute2` in our URLs, exactly as we would do on the internal network. We also don't need to specify a username in the URL, as it is provided by the SSH configuration.

```
$ svn co svn+ssh://ute2/repos/MyProject
```

The working copy is created with a URL that remains valid from both internal and external network connections. Once connected to the internal network, just swap the configuration file to the local version.

```
$ cp ~/.ssh/config_hetlocal ~/.ssh/config
```

And proceed with Subversion operations as before.

Tangentially, this technique can be used to simplify making other secure, encrypted connections from a laptop to computers on the internal network, via SSH, software that uses SSH for its connections (eg. X2Go), or other network software that can be forwarded via specific port numbers (NoMachine, RDP, internal web servers, etc).

Software Deployment

When performing software deployments on HET systems, take heed of some standard recommendations.

- *Installations should be local* to the computer on which they are to be executed. Unless you have a specific reason to do so, avoid installing software to NFS network-shared directories such as /opt/het, /data1, your user directory, etc. Especially in an era of increasing diversity of hardware and operating system versions, this is almost always a bad idea. Put another way, never rely on networked/shared directories (eg. NFS, Windows shares, NetBIOS, cloud storage sync, etc) to do your software distribution for you.
- *Prefer global installations* to a computer for access to all users on that computer, over individual user-specific installs. This implies superuser (root) access to the computer on which the install will take place, a normal requirement when installing software.
- Perform user-specific installs when it is certain the software being installed is of no interest to other team members, and/or when global installations present practical problems or complications that are not easily resolved.
- *Install to standard directories*. In Unix-like operating systems, the traditional directory for installs of most downloaded or locally-developed software is `/usr/local`, in which all software is installed together (without subdirectories specific to particular installs). Executables go in `/usr/local/bin`, libraries go in `/usr/local/lib`, configuration files go in `/usr/local/etc`. In some cases for large software applications, a dedicated directory under `/opt` is created instead. Using `/usr/local` is typically preferred and may on some systems reduce the need to modify binary and library search paths, as it is sometimes included by default.

  Some design conventions are also worth considering.

- Write or configure your software to create a directory for large temporary working files in /tmp. For example, an application called MyProgram, executed by the command `myprogram`, might create `/tmp/myprogram/` for large temporary files. It may be helpful to make this location configurable, but this would make a good default.
- Write or configure your software to save user-specific settings to a hidden user directory or file, according to Unix convention. Using the MyProgram example again, your software could create a directory called `~/.myprogram/` to store configuration details generated while the user is operating the software. Alternately, if it's certain only a single file is needed for such settings, the file could be called `~/.myprogramrc` ("MyProgram resource"). The leading dot (period) character in the file or directory name prevents the directory from being seen by default in graphical views and via the `ls` command.
- Prefer a directory under `/var` by default for large collections of dynamic files generated by your software, especially continuously modified files and named pipes used by multiple programs for communication or coordination. Again, this recommendation may be overridden for specific reasons, left configurable by the user but provided as a default, etc.

The installation procedure is part of your source

The source for your software should contain not only the coded instructions for the computer to execute, but also the instructions to build and install the software itself.

At a minimum, this should include a document describing to a human how to put the software and other needed files in their correct locations such that the software will work as expected. This document should assume as little prior knowledge about the software as possible; a person installing the software in the future may not yet be familiar with its use. Be careful to include prerequisites, other software and libraries that are not part of the operating system and that are needed by your software. If problems are found in your installation instructions later (eg. forgotten prerequisites), edit this document just like any other source file, commit it to the repository, and create a tag for a revised release.

An improved approach would be to automate the install with a script that builds and copies files to their required locations. *This does not obviate the install document described above.* A simple script may involve substantial assumptions about the target system; these should be described in your document.

A further improvement is to implement that automated install in a standardized way, using the GNU `make` utility. Instead of creating a shell script with install instructions, you instead create a Makefile in the root of your project (and possibly others in subdirectories), which the utility will find when `make` is invoked. Inside that file are *targets* which can be specified from the command line, or from other targets. The traditional target name for performing installations would be *install*, such that your software can be installed by the command line "`make install`". Other targets might provide build/compile instructions, clean up unwanted object files from the source tree, etc. A tutorial for creating Makefiles can be found at https://makefiletutorial.com, based on the Make Book written by the utility's authors. This gets fairly deep in the weeds of Makefile syntax, and omits install targets in particular. A tutorial that covers installs can be found here.

The most popular approach for most distributed software, and a good choice if your software is complex and involves a build step and more than a few scripts to install, is to fully automate the build and install process using a platform-independent build system such as GNU Autotools or CMake. The details of doing this are somewhat outside the scope of this document at the present time. As with the above options, the configuration files for these systems are also included in your source, where they can be maintained by version control.

Appendix A - Generating and Distributing SSH Keys

SSH keys are used to replace passwords for authenticating SSH connections. By using keys, the SSH server process can use cryptography to validate that the party attempting to log in to a user account has the same authorization as that user has previously established by setting up the keys.

Public and Private Keys

This is done by mathematically comparing public and private keys. The private key is kept on your workstation (the client side of the connection) and the public key is distributed to your account on servers where you'd like to log in.

The public key is open to public view. There are no security risks to distributing this key broadly; anyone can view it with no consequences. The private key, however, must be protected from access by anyone aside from its owner. This can be accomplished for casual use on private computers with appropriate file permissions.

Key Passwords

On computers with shared superuser (root) access however, complete privacy of the private key file cannot be guaranteed. For this reason, keys should always be generated with a strong password to protect them.

Generate Your Key Pair

The most common method for generating SSH keys is to use the OpenSSH utility ssh-keygen on Unix and Linux platforms.

```
$ ssh-keygen -t rsa -b 4096 -C "My Key Comment"
```

Here, "`My Key Comment`" is an arbitrary string to describe your key. A common approach is to put your email address here, which identifies the key as yours. Including a date string may be helpful as well. This string will be included in the text of the key files.

The "`-t`" parameter indicates the type of key, which implies its cipher algorithm. "`rsa`" is the most compatible choice, being reasonably secure at higher bit lengths and supported on all systems. "`ed25519`" is a much better choice for security and performance on modern systems, but is not supported on older platforms (including `ute2` and most others here). "`dsa`" and "`ecdsa`" are best avoided.

The "`-b`" parameter indicates the bit length of the key for the RSA key type. 4096 is the recommended standard length today. **Do not omit** this parameter; the default is 2048 (1024 on very old implementations), which should now be deemed insufficient.

```
$ ssh-keygen -t rsa -b 4096 -C "cerobison@utexas.edu 2019-11-22"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/chris/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/chris/.ssh/id_rsa.
Your public key has been saved in /home/chris/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:m5cmK6NPzOv10tw1Zh+degR/yLU1hQgaDObzhcxlD2Y cerobison@utexas.edu
2019-11-22
The key's randomart image is:
+---[RSA 4096]----+
|      oo. E. . . |
|     o o.O o. . .|
|      o = . .   .|
|       o .   . .o|
|        S    .oo*|
|     o   o .  B*o|
|      + =o+. +ooo|
|     .oo.*o .. ..|
|    .++o...   .  |
+----[SHA256]-----+
```

Choose a strong password! Save the generated files in the default location (~/.ssh/id_rsa and ~/.ssh/id_rsa.pub) unless you already have keys there and need to choose a different name. The files will be saved with appropriate permissions set; the public key will be world-readable, but the private key should only be readable by your user account.

Distribute Your Public Key

Now that you've created your key pair, it's time to place your public key on servers where you'd like to use it to log in. Let's place it on ute2:

```
$ ssh-copy-id -i ~/.ssh/id_rsa.pub chris@ute2
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/home/chris/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to
filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are
prompted now it is to install the new keys
"Unauthorized use of UT Austin computer and networking resources is
prohibited.
If you log on to this computer system, you acknowledge your awareness of and
concurrence with the UT Austin Acceptable Use Policy. The University will
prosecute violators to the full extent of the law."
```

```
Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'chris@ute2'"
and check to make sure that only the key(s) you wanted were added.
```

The public key data will be copied to the server, not as a separate file, but instead having the text content of the file appended to `~/.ssh/authorized_keys`. You can edit the `authorized_keys` file manually by some other means (eg. an editor like `vim` inside an SSH connection) and paste the key contents into this file yourself; `ssh-copy-id` does the job in a single command and sets file permissions appropriately.