Team Working With Shoebox

Martin Hosken, SIL International

Introduction

- Multiple people
  - Same project files
    - Database files
    - Settings files
- Remotely located
  - No direct connection
- Expertise Centralised

A typical remote project involves a small team of people working remotely from each other. The need is the ability for these people to share files that they are working on with each other. The complexity comes from the need for these people to be able to edit the various files they are working on and then to collect those changes into a central repository that holds the master form of the various documents.

In the case of a Shoebox project, the files involved are both database files and the Shoebox settings files. The settings files should be included in a project to account for changes to database schema, and language settings. Notice that the .prj file should not be included.

Two aspects of the problem which make it somewhat unique, in addition to the nature of the files being shared. The first is that the team is so remotely located that there is no way for the computers involved to make direct contact. Communication of data is via email, mail or carrier pigeon.

The other aspect is that very often the expertise for managing such a project is located with only one or two members of the team, rather than all team members being proficient computer users.

Introduction: Code Management

- Multiple Users
- Text based files
- Direct contact with repository
- High level of user expertise

There are existing tools and techniques that have been worked through in the software development world. Software is often developed remotely with users in various parts of the world. Most software projects are developed using such tools and techniques.

It would be nice to be able to use such tools directly, give some training and then go home. But unfortunately this isn't entirely possible.

First while Shoebox files are text based files, the text based files that get put into a code management repository are considered to be line based rather than record based. Thus line ordering is very important and if two lines are inserted at the same place, then this is considered a clash, whereas as adding two records to a database at the same place is not considered a clash.

Second, these tools presume that a client has direct electronic access to the repository. There is some hope that this may change in due course, but to date, there is no clear way of resolving this one without a lot of work.

Finally, such tools are used by programmers who have a high level of computing expertise,and even then they can have trouble getting their heads around this software.

Introduction: Approach

- Scenarios
  - Introduce principles
  - Provide cookbook examples

The rest of this discussion will present various scenarios and how they are addressed. We start with some basic, somewhat unrealistic scenarios to introduce the basic principles and then slowly grow into the complexities of real world projects.

A secondary aim of these scenarios is to provide a cookbook of examples that can be used later by users.

Scenario 1: History

- How do I keep a history of particular versions of the project files?
- Copies of the directories
  - .zips of the directories
- Version naming
  - *project1.zip, project2.zip, ..., projectn.zip*

We start with a simple question.

There are various ways of solving this problem. Can you think of some?

Others include keeping a floppy or CD per version and keeping the files off the machine.

The .zip approach gives a good solution to the problem. They are an easy way to handle multiple files, and it is not as if we need to run any programs on old versions of the data directly. We can always unpack them if we so desire.

The version naming scheme is going to be important, so bear with us as we introduce it.

Scenario 2: Patching 1

- My friend sent me some files and I have made changes. How can I send the changes back?
- Send a copy
  - Probably as a zip
- Send only the changes
  - Need to keep the original files we received
  - Need a program to extract the changes

I am a remote user and someone sends me a set of project files to work on. I have made some changes and now I want to send all my hard work back to the original sender. What should I do?

Again, there are many approaches to this problem.

The first approach is to simply send a copy of your files back to the sender. We would probably send and receive .zip files.

But what if the files get really big. For example, what happens if my 5MB dictionary only has a few changes in it? Do I have to send it and all those texts back? Can't I just send the changes I have made?

In order to do that, you need to keep a copy of the original files you were sent. Probably these still reside in the .zip file you received (if you even detached from your email client). And then you need a program that will compare the files you have now with those you were sent. You can then send the report from this program back instead of all the files.

Scenario 2: Patching 2

- I have just received a *difference file* from a remote user. What am I supposed to do with it?
- Recreate the files the remote user has
  - Need the original files I sent him
  - Need a program to apply the differences to that file collection to create the files the remote user has

At the other end of the email link is someone with the opposite problem. They have just
received a difference file from us. What are they supposed to do with it now?

Obviously, they need a copy of the files they sent to the remote user. Notice that this is not necessarily the same as the set of files they are working with at the moment. Thus it is important that they keep a copy when they send them out to remote users.

In addition, they need a program to apply the differences to those files they sent out, to create the same set of files I have here.

Scenario 2: Solution

- Creating the *patch*:
  - `zipdiff proj1.zip working > proj1.patch`
- Email `proj1.patch`
- Applying the *patch*:
  - `zippatch –o remote proj1.zip proj1.patch`

The first command takes an original set of files and compares it to a changed set of files. It outputs a file that will update the first set of files to be the same as the second set.

We then email this patch file back to base.

The second command takes a patch file (as they are known) and applies it to a set of files (in this case held in a .zip file) and outputs the updated set of files to the remote directory (it could just as well be remote.zip, a .zip file).

Scenario 3: Merging changes

- We've both made changes to our files. How do I merge our changes?
- Not all changes are in the same place in the file
- Compare files and select which changes to keep
  - Tedious

Now comes the hard part. You have made changes and so have I. What do we do now?

I could go through the two files by hand and decide which differences in each file to keep. But this is an inordinate pain. Imagine doing that for all your files you have just re-interlinearised!

What is needed is some kind of tool that can do this automatically.

Scenario 3: Solution

- `zipmerge proj1.zip remote local proj2.zip`
- Takes changes to original files and merges the changes into new originals

original	A	B	output
x	x		
	x		x
x	y	x	y
x	y	z	!
x	y		

zipmerge is a program that takes three directories and compares them. For each file in the given directories, it compares them against the originals. Thus we could say that for a particular file set we have:

merge original filea fileb result

The program then looks at the differences between filea and the original and also the differences between fileb and the original and works out which changes to apply. For the most part it can distinguish between non-interacting changes for filea and fileb. But if both filea and fileb edit the same part of the original, then it may result in a clash.

For each line in the file, the table shows what happens for each case. In the last case, it depends on the particular merging algorithm in use. For Shoebox both changes are included, but normally this is considered a clash.

Scenario 3: Shoebox merging

- Records in any order
- Multiple records with same key field
- Field order significant

original	A	B	output
x	x		
	y	z	y z
x	y	x	y
x	y	z	!
x	y		y

If zipmerge knows that a particular file is a shoebox file, then it can do some special processing to improve the merging process. It knows that records may occur in any order. It also tries to line up multiple records with the same key field. It does this by looking at the other fields in each record and working out which records line up with which based on the fewest differences in fields. Finally if there is a clash in two records being edited, it does a field by field (like line by line) merge of the contents of the records. Thus there should only be a clash if two users edit the same field differently (i.e. if they both insert different fields, that is no problem).

Notice that where a record is both deleted and edited, we keep the edited version rather than delete. This is so that we don't accidentally delete data that someone wants to keep. We assume if someone edited a record that they intend that it be kept.

The chart shows what happens in terms of records in the database. A clash is resolved by doing a field order (like a line based) merge on the records. Thus only clashes at the field level cause a clash to be output. Notice that if two records with the same key field marker but different contents are added, then both records are added to the output.

Life gets interesting when people start editing key fields. Editing a key field is the same as deleting and inserting a record. If someone else edits the record that has effectively been deleted, we want to keep it and then allow the clash to be sorted out later, if it is noticed!

Scenario 4: Manifests

- How do I only work with certain files?
  - No binary files, .prj, .bak, auto-generated files
- Use a Manifest file
  - Contains a list of files to work on
    - .lng, .typ, .db, .txt, etc.
  - Also gives the type of the file
    - `mydict.db text/shoebox`
      `settings/lexicon.typ text`
      `settings/Default.lng text`

If we just bundle up directories and pass them around, even as .zips, there is still the problem of .bak files, and other redundant files. We don't want .dlls and other binary files passed. The only files we want to pass around are those that a user edits (either directly or via Shoebox, etc.)

The answer is to use a manifest file. Such a file contains a list of all the files that we are interested in. It also has the side-effect of being able to include the type of each file. This is particularly important because how else is zipmerge going to know that a file is a Shoebox file?

Due to the way the programs work, filenames and directories in the manifest file are CASE DEPENDENT. In addition the filename and type must be separated by at least 2 spaces or a tab. (This is to allow filenames with spaces in them. I am assuming that people don't work with filenames including multiple adjacent spaces).

Scenario 4: Solution

- `zipmerge –m manifest.txt orig a b result`
- `zipdiff –m manifest.txt orig new > patch`
- `zippatch –o new orig patch`

Each of the tools is able to work with a manifest file. zipdiff only produces differences between files listed in the manifest. Notice that it is a good idea to include the manifest file in the manifest so that it is updated when returning results. zipmerge only works with files in the manifest and also merges the various manifests.

zippatch is odd in that it doesn't need a manifest. Why?

Because all the files that need changing are in the patch file.

There is no problem passing more files around than are in the manifest. So an initial .zip could contain many more files than are in the manifest. From then on, patches, etc. would not cause changes to those files that came in the original .zip

Scenario 5: Remote changes

- I am a remote user. I sent off my changes yesterday, but have made some changes to the files before I received my update. What do I do now?
- Use zipmerge to merge the updates into your working files:
  - `zippatch –o temporig.zip proj1.zip proj1.patch`
  - `zippatch –o tempnew.zip temporig proj2.patch`
  - `zipmerge –m manifest.txt temporig.zip tempnew.zip working new`

This gets a little complicated (which is a shame because this needs to be done by a remote user). First we need to make a copy of the files that we had when we sent off our patch. Then we need to make a set of files that we have received. Finally we merge the received files and our current files against the files that we sent off. This gives us a new working area.

If anyone is feeling brave they could change the last command to:

zipmerge –m manifest.txt temporig.zip tempnew.zip working working

But that would take bravery!

Thankfully it is possible to hide much of this in a batch file.

Scenario 6: File naming

- This is confusing. How do we keep track of it all?
- Careful naming of files:
- Main repository:
  - *proj*1.zip, *proj*2.zip, ..., *projn*.zip
- Patches:
  - *proj*1_2.patch, *proj*1_*user*.patch
- Temporary copies of changes:
  - *proj*1*user*.zip, ...

Since files have to be around for a while, it is worth getting the naming scheme right. There are many naming schemes, which can borrow from code management systems. We introduce the simplest here:

Versions in the main repository are .zip files with version numbers on the end of the file. These are the centralised merged history copies that get distributed to folks, either as full .zips or as patches.

Patches are given the name of the .zip that they should be applied to. Thus if I were returning a patch to base, I would use the name of the file in the repository to apply this patch to. Likewise if someone were sending me a patch they might apply it to a previous main version, or my update to a version.

Personal updates to a version are given the version number of the base version and the user name/abbreviation for the user.

Scenario 7: Multiple Updates

- What happens if someone in a team is very slow in replying?
- Merge their changes with the current latest version

I send out version, say, proj7.zip and various other members of the team send in their changes and I update the main repository accordingly. I am now at proj10.zip. Then a slow member of the team, who hasn't been keeping up, sends in their changes, but against proj7.zip. What do I do now?

zippatch –o proj7slow.zip proj7.zip proj7.patch

zipmerge proj7.zip proj7slow.zip proj10.zip proj11.zip

But this shouldn't happen because people should be updating their working copies from the updates you send out each version.

Scenario 7: Summary

- Base sends out latest update patch: `proj6_7.patch`
  - `zippatch -o proj7.zip proj6.zip proj6_7.patch`
- Users *fast* and *slow* make changes.
  - `zipdiff -m manifest.txt proj7.zip working > proj7_fast.patch`
- Base receives the patch from *fast*:
  - `zippatch -o proj7fast.zip proj7.zip proj7_fast.patch`
  - `zipmerge -m manifest.txt proj7.zip proj7fast.zip working proj8.zip`
  - `zipdiff -m manifest.txt proj7.zip proj8.zip > proj7_8.patch`
- User *slow* receives the new patch
  - `zippatch -o proj8.zip proj7.zip proj7_8.patch`
  - `zip working proj7_slow.zip`
  - `zipmerge -m manifest.zip proj7.zip proj8.zip proj7_slow.zip proj8_slow.zip`

Phew!

First we assume that everyone is at rest. Thus nobody has any changes after proj6.zip. The patch is sent out and everyone applies it to create a new proj7.zip which they can unpack and work with.

User *fast* gets lots of work done and sends in some changes as a patch.

Base receives the patch and creates a temporary version of what user *fast* saw when they made the patch. Then, if they have made any changes (or have received changes from others), they merge in those other changes to create a new proj8.zip

Finally they send out a patch to update everyone from proj7.zip to proj8.zip

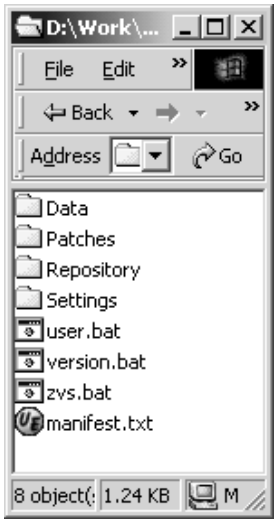
But user *slow* hasn't sent in their updates and they suddenly receive a patch to update their proj7.zip. But they have changes against proj7. So what are they to do?

First they create proj8.zip then they create a copy of their working directory and merge that along with proj8.zip to make a new update to proj8. They could just make a patch of that and send it in, or they can unpack it and carry on working before sending in.

This is about as bad as it gets. If you don't use zippatch and zipdiff, then things get easier and only the zipmerge commands are needed.

Scenario 8: zvs

- This is *so* complicated. Can we not at least make it easier for remote users?
- Simplification implies fixed structure
  - ./Repository
  - ./Patches
- Not necessarily easier for repository manager



This is very difficult to work with, with all these command lines. Is there anything we can do to make it easier for remote users who are not necessarily highly computer literate.

Yes, but the cost comes in making more presuppositions about the file naming scheme and the directories, etc.

For this system, we assume that all the files are held below some common directory which we will call the root. Under this we have two directories: repository, in which we keep all the versions of the project as projn.zip, and patches into which users are encouraged to place .patch files they receive, although they can also go into the root directory. We presume that there is a manifest.txt that is being used, and most of all we presume that the repository manager makes no mistakes!

Notice that the tool we have here is not going to make life easier for the repository manager, yet.

Introducing zee vee ess, which is designed to make the creation and management of patches and zipmerging a whole lot easier for remote users. It is written purely in DOS Batch, which makes it a rather limited program. At the moment it supports two commands: send and receive.

zvs send creates a patch to send to the repository. It automatically knows which version to zipdiff against and also the user id of the user involved. So it can create an appropriately named .patch file.

zvs receive takes a .patch or .zip that has been received from the repository and merges it into the working area of the user. The .patch file should be stored in the Patches\ subdirectory or alternatively in the root directory. The .zip file must be stored in the Repository\ subdirectory (if created by a .patch, that is where it will be stored).

Scenario 8: zvs

- Zip Versioning System
- `zvs send`
  - Creates a patch to send to repository
- `zvs receive`
  - Installs a patch from the repository
- Presumes:
  - manifest.txt
  - Version.bat
  - User.bat
  - zvs.bat

How does zvs know which version number we are dealing with and who the user is? The answer lies in two files that zvs calls. Version.bat contains three set commands:

set proj= project name prefix for files – e.g. proj

set cv=current version number (i.e. the number in projn.zip for this .zip file)

set nv=cv + 1 (since batch language can't do maths on a Win98 machine ☹)

Notice that it is essential that the respository manager keep this file up to date. For this reason, this file should be included in the manifest.txt

User.bat, on the other hand, is private to the user and contains a single set command:

set uid= user id for use in patches, e.g. set uid=mh

Since this is file is unique to each user, it doesn't want to be propagated to everyone. Instead the user will have to edit it once at the start of the project and then forget it. But it must not end up in the manifest.txt

Finally, zvs.bat itself is a program that is running while a project is being merged into the working area. Therefore it cannot be written to (since it is open for reading when the merge occurs) and so should not be part of manifest.txt. Changes to zvs.bat should be sent separately.

The requirement is also that remote users keep up to date with the patches they receive. zvs does not support jumping versions.

Repository Manager

- Creates initial release
- Converts user patches to .zips for own use
- zipmerges everything and resolves clashes
- Updates Version.bat and manifest.txt
- Creates and sends out patch from n-1 to n
- Never makes mistakes!!

The repository manager needs to be aware of and able to use all the various zipdiff, zippatch and zipmerge tools. Their role is to receive updates from users and to merge them into a master update that is sent back out. Where possible, they should handle clashes or at least be aware of the clashes and work with people to ensure they are resolved before causing wider problems.

Very often a repository manager is also a user, and it is very advisable that repository managers try to keep the two roles separate by keeping a very separate directory structure for their repository manager role and then to use the same structure as remote users for their normal working role.

One weakness with the system is its lack of resilience towards mistakes that may creep in to files due to patches being created the wrong way around or against the wrong version of a file. Errors can easily creep in if version.bat is not kept up to date.

Conclusion

- Easy to use for clients
- Hard on the repository manager
- Brittle (any errors propagate badly)
- Need a proper management system
  - Handles multiple file formats: SH, XML, binary, text
  - No central repository
  - Not so brittle
- A good start

In conclusion, what do we have?

We have a tool that is relatively easy to use for clients, assuming that it is set up well and that users don't make too wild mistakes.

Unfortunately, the setup as it stands is somewhat brutal towards the poor repository manager who is expected to be perfect. This I would consider the biggest weakness of the system as it stands.

The system is brittle. Any errors that may creep in will be hard to get rid of. This is another major weakness which is inherent in merge based systems since such systems tend to encourage changes and so keep errors. But, since you can go back to any previous version of the setup and them merge forward from there, it is not so bad if someone is aware and can regain control.

What is needed is another layer on top that is a proper management system. What we have is a good start and example of such a system, but it needs all the other helps that such a system provides. It needs to handle version numbers for you and make sure that everyone knows which version they are working with, etc. It needs to be easy to set up (which this system is). But what is really needed in a future system is the ability to understand different file formats and to merge accordingly, for Shoebox, XML, binary, line based text files, etc. In addition, the lack of a true central repository (in effect each client has their own repository and there is a process of keeping those repositories in mainline synchronisation is the hard work of the repository manager) is a unique requirement for this system. What we have with zvs is something that does the job, but has weaknesses. And obviously something that is less brittle, that doesn't let users do things wrong (or makes it hard for them to do things wrong).

Grade? A good start