Managing Source Code in a Distributed Environment:

New Views on Some Old Problems

David Tilbrook
Dean Thompson
Mark Lorence

Carnegie Mellon University
dt@andrew.cmu.edu
dt@cs.cmu.edu
mark@andrew.cmu.edu

ABSTRACT

The management of source code is difficult, even when done on large shared central machines. The use of multiple machines joined by a network does not introduce many new problems. Rather it magnifies weaknesses of the single machine solution (such as it is) and increases the importance of taking proper measures to ensure that the source is protected and well-managed. The objective of this paper is to explore the problem domain, uncover specific problems, and present an approach to maintaining a body of source code when development is done in parallel on multiple machines.

1. Introduction

When the programmers who develop a medium or large software product are spread across a heterogeneous network, controlling the quality and consistency of the code they produce is exceedingly difficult. Source and quality control was never easy, and has never been done exactly right, but it seemed much more tractable a few years ago when most products were built on large central systems. Distributed development has not introduced many fundamentally new problems. Rather, it has magnified the old ones so much that a new approach is needed. There are two major reasons the old problems have been aggravated. One is the presence of heterogeneous hardware. To say that two programmers were working in ‘‘different environments’’ used to mean that one was using a VT100 terminal and the other a VT102; now one is using a RISC architecture uniprocessor and the other a multiprocessor Vax. The other new factor magnifying old problems is that development occurs at geographically distant sites. Two programmers who were ‘‘not in close physical communication’’ used to be five flights of stairs apart; now one is in New York and the other in San Diego.

In this paper we will concentrate on the problems introduced when development groups are separated and communication is limited. Traditional source control systems like RCS and SCCS assume that all developers share a common source database. These systems do not work when file transfer is too slow for a single source database to be practical. Also, conflicting changes become common when people are physically separated, and the conflicts are much more difficult to detect and resolve. In addition, programmers at distant sites must often test their updates using different versions of the system in different software environments. The existence of many versions of the product is no longer an error to be avoided; it is now a reality that must be accommodated.

By struggling with several large distributed projects, the authors have developed an organizational approach that we believe best solves these problems. The remainder of the paper describes this structure and explores the information flow within it. The exploration focuses on the special difficulties that arise in a distributed environment, and draws on experience to suggest general approaches to overcome them.

2. Basic Goals

Before discussing solutions to these problems it is important to agree on exactly what the goals are. Why do programmers at different installations collaborate? Successful software management strategies become vitally important only when the collaboration is directed toward producing a single reliable *product* of substantial size. This product will not necessarily be marketed, but it will be used for a long time by many people besides the developers, in many different versions, and must be supported. The policies, procedures, and tools for handling the product's source code should smoothly support a variety of needs during its lifetime.

The most important goals for any software development organization include those listed below:

Delivering the system to a client: Someone must eventually produce a consistent version of the system, test it, repair errors that are found, and ship it to one or more clients. After a version is delivered, there must be a clear trail relating it to the development path, and a systematic way to reproduce the delivered version later.

Testing: Most bugs are found by developers testing new code. Many others are found during testing before a system is released. When a bug is found by one site and must be fixed by another, reproducing the bug is often difficult.

Fixing bugs found by a client: A bug (ok, maybe several) will inevitably be found in the version of the system that is delivered, and must be fixed.

Preventing incompatible changes: Developers need ways to avoid simultaneously changing the system in incompatible ways. When only a single site is involved, a simple version control system can prevent simultaneous changes to individual files or modules, and informal communication can prevent most other types of conflicts. If a common source database is impractical and the developers are not in daily contact, these simple techniques break down.

Resolving incompatible changes: Whenever two or more programmers build a system together, they will occasionally introduce conflicting changes no matter how closely they communicate. When many programmers cooperate at a distance, incompatible updates are common and must be dealt with routinely.

These goals are formidable. There is no hope of achieving them unless software management is accepted as a priority by everyone on the project, from the beginning. However, let's face it, people will only cooperate when it is convenient for them. Explaining the abstract benefits of software management will motivate project members for about an hour; demonstrating that your set of mechanisms is easy to use will motivate them forever. Simplicity is vital. Every time the software management system interacts with an individual whose first concern is not software management it must clearly help them rather than inconvenience them.

3. An Organizational Model for Distributed Software Management

To facilitate the discussion of the goals described above, this section will provide a conceptual framework for our analysis and introduce some terminology. These terms are, for the most part, conceptual. Where necessary, stronger or more explicit definitions will be given in later sections.

3.1. The True Source and One True Source

Ideally, it should be possible, upon demand, to create a collection of consistent, correct source files that represents the best currently available version of the product. Such a collection is defined as a **True Source**. Its importance should be obvious, since it is the foundation for products delivered to the end-user and, normally, the foundation for any future modifications or enhancements.

The production of a **True Source** is so difficult and expensive that it should be done only when absolutely necessary for delivery to a client. For development, the requirements are not as severe.

The most up-to-date source collection that meets some minimal criteria (including some quality assurance by the most recent contributors) will be referred to as a **One True Source** (OTS) [Tilbrook/Lord 87]. It is used for development, and as the initial collection of source files that becomes a True Source through further quality assurance.

3.2. Developers, the Gatekeeper, and Clients

The **developers** are the people who create or modify source code for the product. Sometimes such people may fulfill other roles with respect to the product (e.g., testing and integration, end-user support), however, the term is used to refer to their role as source creators only.

In practice, it has been our experience that achieving an OTS is impossible, unless someone assumes and fulfills the responsibility for creating it. This is particularly true when the developers are not using a shared file system, and thus create multiple copies of the source. Furthermore, to create an OTS, it is essential that the developers recognize the role and importance of such a person and endeavor to provide the necessary information and assistance. The individual or group that maintains an OTS is henceforth referred to as a **gatekeeper** [Tilbrook/Stern 87].

We assume that software is being developed to be delivered to people who commissioned, purchased, have agreed to use, or are responsible for further processing of the product. Such people are referred to as the **clients**. Note that clients are not necessarily end-users; they might be members the same organization who are responsible for further processing prior to final distribution.

3.3. Communication Channels

By placing the gatekeeper at the center of a star-shaped figure, three main communication channels are created (see figure 1):

- the **gatekeeper-client** channel
- the **gatekeeper-developer** channel
- the **developer-developer** channel

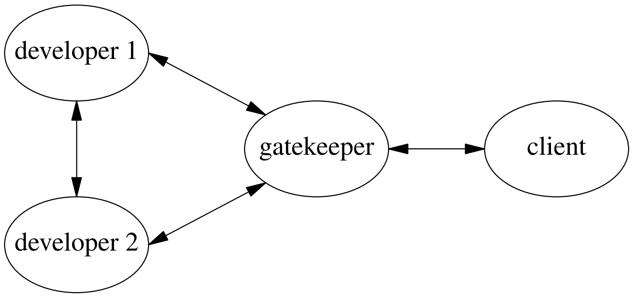

Figure 1

We are using the term **channel** to describe a path for information flow. Each channel must be bidirectional and "versatile," meaning able to convey voice, paper and electronic information. In later sections, we will discuss exactly what information flows along each channel.

4. Problems Handled by a Software Management Organization

Previous sections of this paper examined the goals and difficulties of distributed software management and outlined a basic organizational model for addressing them. In this section we look at some specific problems the organization must handle and discuss what information must be passed along each communication channel. We also begin to consider exactly what databases each group in the organization will need to maintain.

4.1. Maintaining the One True Source

Because so much complex information must be combined from different sites in order to build an OTS, we believe the only reliable way to ensure that an OTS can be constructed is to continuously maintain one.

Developers must be able to submit their new source to the gatekeeper who, after testing and other quality assurance, incorporates the updates into a new version of the OTS. One way to accomplish this is to permit developers to create new entries in an RCS or SCCS administration file, and let them specify which versions the gatekeeper should extract for inclusion in the OTS.

4.2. Synchronization Between Developers

The versions of the system source being used by developers must be kept synchronized to some extent. We believe that maintaining complete agreement of source at all sites is both impractical and unnecessary. Developers at different sites can work productively on different versions of the system as long as they are not missing extensive updates to areas of the source that concern them. They must be able to determine what updates they need and receive these updates reliably (which means mechanically) and on time. Developers need to receive complete copies of the source when they first come on-line and again, infrequently, to protect against accumulated discrepancies.

Largely because of the demands of this synchronization process, more information flows along the gatekeeper-developer channel than any other. Updates must be transmitted promptly, and will sometimes be extensive. Information about active development areas will be updated and accessed frequently by all development sites.

In some cases, it may be useful for two developers to bypass the gatekeeper and communicate directly between themselves. The developer-developer channel allows developers at one site to ‘‘query’’ those at another about the status of particular modules, or to receive small, timely updates without involving the gatekeeper. This path must be watched carefully, however, and should be replaced by the standard gatekeeper channel whenever major updates to the system are required.

4.3. Conflict Avoidance and Resolution

One of the most irritating and dangerous problems of loosely coupled development is the possibility of conflicting changes being made to the same source file. Conflicts can also occur within modules or between different modules.

A variety of techniques can help avoid such conflicts. Usually it is possible to divide the source tree into areas of concern. This reduces the number of people who might conflict in any given area, and hopefully places them all at one site. When a group of physically separated developers must work on closely related source, the developers must carefully report new activity in that area both to the gatekeeper and to the other sites involved.

Because the above procedures will inevitably fail in some cases (remember Murphy), there must be a procedure for detecting and resolving conflicts that occur. One possible mechanism is as follows. Whenever the gatekeeper receives a modification to source, she confirms that the change is to a file that was last modified by the submitter, or that the version of the file from which the new source descended is the most recent in the OTS. If both of these checks fail, the gatekeeper may refuse to accept the change and demand that the submitter merge his changes into the most recent version of the OTS.

4.4. Testing and Bug Fixes

Ideally, most of the testing for each update to the system will be done by developers before the update is incorporated in the OTS. The developers, however, cannot do final testing of the entire system, because they have both a different version and a different environment than the gatekeeper. The major purpose of testing done by the developers is to increase the number of bugs that are found by the site which introduces them, and reduce the number that are found by the gatekeeper or other development sites.

When a bug is found at a site other than where it was introduced, the bug report must include both data about the environment at that site and enough information to reproduce the erroneous version of the system. If it is clear which part of the system is in error, and if a particular site has responsibility for that part, the report can be sent directly. Otherwise, it should be sent to the gatekeeper. Bugs found by clients will always be sent to the gatekeeper.

Once responsibility for a bug has been assigned, and a solution found, the OTS must be updated. If the bug is serious, its fix can be sent as an update to the affected clients. Ideally, a work-around should be devised and sent at least to the client who reported the bug, and possibly to others. The gatekeeper must ensure that developers know of the bug and the fix, so they will not later reincorporate the same bug into the OTS.

4.5. Integrity Checks

No matter how careful the developers or fastidious the gatekeeper, errors and inconsistencies will occur in the construction process. Problems will arise for two basic reasons: environmental differences and errors in information transfer. Environmental differences that can cause problems include differences in CPU type, peripherals, operating systems, and supporting software. Information transfer errors mostly stem from inevitable oversights and miscommunication by both developers and the gatekeeper.

It is essential that the gatekeeper and the developers institute a set of procedures that periodically check the integrity of the system. Problems caused by environmental discrepancies can be caught if the gatekeeper completely reconstructs the system in a clean environment. Errors due to human mistakes and miscommunication can be culled out through routine validation of the database and periodic comparisons between the OTS and versions of the source maintained by developers. Each of these checks deserves a brief explanation:

- **database validation**: The OTS must be regularly checked for inconsistencies. One useful technique is to create a list of all files actually in the system and compare it against a list of files which has been maintained by hand. Each file in the OTS should be accounted for and compared against a new version extracted from the gatekeeper's database.

- **full reconstruction**: Periodically, after the list of source files has been carefully checked, these files should be transported to an isolated environment which matches the ‘‘vanilla’’ client system as closely as possible. The system can then be built from scratch and tested. This can be done for each different target environment, and dependencies on the development environment are likely to be caught.

- **comparison between versions**: Occasionally, complete comparisons should be made between different versions of the system. The OTS should be compared against developers' versions of the source to be certain that no local changes have been lost. Different released versions of the system can also be compared to see if all updates can be accounted for. Clearly, the amount of manpower available will influence the thoroughness of these comparisons.

The above list is by no means complete. Just the major checks are listed. Most importantly, the gatekeeper should firmly believe that Murphy was an optimist, and be constantly concerned with ensuring the integrity of the system.

5. Information Maintained at Each Site

At this point we can summarize the information needs of the gatekeeper, the clients, and the developers. These needs are extensive enough that we believe a good integrated database (including a comprehensive query facility) would be an excellent foundation. In this section we will often describe specific lists as though the information were kept in a separate file, because this is how we have kept it. Ideally, a good database would make some information easy to generate at need.

We want to present a careful, organized approach that scales up well. It is a fact of life that large systems grow by accretion. New developers become involved, new extensions are devised, and medium sized projects meld with each other and with larger ones. Any source management system that does not mature gracefully can expect a long, painful, tottering old age.[1]

[1] Any reference to the principal author is purely coincidental.

5.1. The Source Database at the Gatekeeper Site

We use the term ''source'' to cover every file that is created and modified by hand, rather than mechanically generated. This means more than just those files that are fed into compilers. Documentation (especially internal documentation) and construction information (such as makefiles) are equally important.

The gatekeeper uses a version control system to manage the source. Without describing the details of such a system, we claim that a good version control system should keep relatively general ''deltas'' describing the changes that convert from one version of the system to another. A delta describes updates to specific files, deletion or creation of files, renaming of files, and even renaming of directories. General deltas of this sort are vitally important (and we have sorely regretted not having them) so that one site can transmit just the list of updates needed to produce a local version of the system from one available at other sites. Separate audit trails are essential to this process. Unfortunately, most common version control systems lack this feature.

Version numbering when there is distributed, parallel development presents a unique set of difficulties. Many different (imperfect) solutions are possible, and a complete discussion is far beyond the scope of this paper. We briefly describe one possible solution where the gatekeeper and developer are only allowed to edit files from their respective source trees. (A file ''transferred'' from one developer to another becomes the responsibility of the receiving developer.) We suggest that each version of a file contain the version number of its most recent ancestor in the OTS database, the name of the development site that changed it last, and the version number of its most recent ancestor at that site. A clear marker beside the two version numbers should indicate whether the file was most recently modified by the gatekeeper or the development site. A database can be used to preserve the mappings between the gatekeeper and development version numbers. The reader should be warned that we have never used this version numbering scheme, but we think it would be a good one.

5.1.1. The Source Database at the Developer Site

The local development sites need a method of tracking changes to their individual systems. A standard source control system, like RCS or SCCS, can be used as long as the developers realize that their source must eventually be transmitted to the gatekeeper. The developers must also keep track of what they send to the gatekeeper. This will provide a version number check. For example, the gatekeeper may report a bug as follows: file ''parser.c'', OTS version number 1.3, developer version number 1.17. The developer can then check to make sure that his version 1.17 is indeed what he sent to the gatekeeper to become OTS version 1.3.

5.1.2. List of Activities

The gatekeeper should maintain a list of ongoing activities to reduce the number of conflicting changes that occur. Ideally, the list should start with a high-level segmentation of the entire system and an assignment of each segment to a small group of developers. In practice, the segmentation will usually be blurred and some portions of the system will be entirely public. This list is updated frequently to reflect the current activity at each development site.

5.1.3. System Snapshots

The gatekeeper must keep a snapshot of every delivered version of the system, until that version is made obsolete either by an explicit contract with each client who received it or by a general sunset clause. In principle, the source code for any version could be generated from a list of version numbers and the source database, but it is safer to explicitly keep exactly what was sent out.

5.1.4. Bug database

Much has been written about the design and implementation of bug databases. In a distributed environment, the important point is that a system-wide bug database should be kept by the gatekeeper. Additionally, each development site will most likely have a database specific to its section of the project. The gatekeeper must be informed of all bugs in any version of the OTS.

6. Summary

This section offers some guidelines for implementing the design discussed in the previous sections. For the most part these guidelines relate not to the systems used, as much as to the attitude and human approach to the problem domain.

6.1. The Gatekeeper

We believe the key to successful software management is to have a single person, who we call the gatekeeper, primarily responsible for the integrity of the software source database. Because this job is difficult, the gatekeeper must have a strong belief in the value of software management.

6.2. The Attitude of the Developer

Successful software management requires the sympathy, understanding, and cooperation of each developer. At times, the developer may feel burdened by the gatekeeper's requests. In reality, the gatekeeper is performing a tremendous service to the developer by dealing with large numbers of files from many suppliers. Developers can assist the gatekeeper by using good programming practices (modularity, standards, etc.) and by performing thorough testing.

6.3. Formalized Information Structure

In distributed environments, a formalized and well-defined structure for the project information must be specified and understood by all. In large, distributed projects, special cases (e.g., special flags to the compiler, arcane source pre-processing) should be avoided as much as possible. Naming schemes and other standards should be defined and adhered to strictly. The gatekeeper is often required to deal with the system at a course level of granularity. She cannot afford the time to learn or deal with gratuitous differences across sub-system or directory boundaries. Eliminating special cases substantially reduces the volume of information that must be maintained, and simplifies the automation of tasks.

6.4. Organizational Priority

Finally, and perhaps most important, the organization itself must give high priority to obtaining sufficient labor and capital for software management. Project planning and scheduling must incorporate release engineering. Project designs and specifications should include the source management procedures and standards to be employed by the project team. Software management should never be an afterthought.

7. Inconclusions (note to ed. this is spelled correctly)

Early in the development of this paper, the authors concluded that this was going to be a difficult one to write. Software management is an area long on problems, and short on solutions, even in the homogeneous system environment. In discussions with the publisher, we concluded that the overall reaction to this issue would be to make the reader lie awake at nights worrying about the problems posed. We think readers may have some of the following initial reactions.

I thought this issue was supposed to be about networking? – Again, we believe the major problems are not really due to networking. Distributing the development of software over multiple machines mostly amplifies problems that were present in well-integrated environments.

These solutions seem to depend excessively on a competent and conscientious gatekeeper. – Software management is a difficult and poorly understood problem. We felt it essential to use strategies and policies that were simple, feasible, and implementable. We are fearful of imposing autocratic or unacceptably complex mechanisms. A human gatekeeper adds the intelligence and flexibility that is needed to make the system usable while still highly effective.

We (the readers) have a better approach that avoids the problems you solve. – The authors are aware of some of the Integrated Programming Support Environments that are now available (e.g., IST's ISTAR [Dowson 87]) and believe that ultimately such systems will solve or avoid many of the problems that were presented in this paper. However, we feel that such software is beyond the means (both financially and technically) of most organizations at this time.

It's Utopian B.S. – (Or, more politely, *software management can't be worth this much work.*) It is our belief and experience that careful software management during development is far cheaper than the release engineering required without it. It is also much less costly than the client dissatisfaction that will result if release engineering is done poorly.

Well understood and effective software management greatly increases the productivity and effectiveness of the developers and the support group. The approach described here has been used [Tilbrook/Place 86], and works, and is now serving as the basis for our future research plans. Work on a more ambitious and comprehensive distributed software management system is being undertaken as the required new technology becomes available.

References

[Tilbrook/Lord 88] D.M.Tilbrook and Tom Lord, *Polices and Tools for Hierarchically Managed Source Code Development*, to appear in Computing Systems, 1988.

[Tilbrook/Stern 87] D.M.Tilbrook and Z.Stern, *Cleaning Up UNIX Source -or- Bringing Discipline to Anarchy.*, EUUG Dublin Conference Proceedings, September 1987.

[Dowson 87] Mark Dowson, *Integrated Project Support with ISTAR*, IEEE Software, Nov. 87, Vol. 4

[Tilbrook/Place 86] D.M.Tilbrook and P.R.H.Place, *Tools for the Maintenance and Installation of a Large Software Distribution*, EUUG Florence Conference Proceedings, April 1986. USENIX Atlanta Conference Proceedings, June 1986.