VC^2 - Providing Awareness in Off-The-Shelf Version Control Systems

Daniel Machado[1], Nuno Preguiça[2], Carlos Baquero[1], and J. Legatheaux Martins[2]

[1] DI/CCTC, Universidade do Minho
[2] CITI/DI, FCT, Universidade Nova de Lisboa

Abstract. Version control systems have been used to help groups of people working at the same or distributed sites to cooperatively create documents. In particular, these systems are very popular in distributed collaborative software development. However, even using these systems, users often perform concurrent changes that require manual conflict resolution, as it was confirmed by a study on conflicts in cooperative software development. Important causes for this situation are the lack of mutual awareness and coordination, among developers, and reluctance to commit unstable modifications. The paper addresses this problem by providing a tool that integrates with off-the-shelf version control systems and monitors filesystem accesses to relevant files in order to enhance the awareness among developers. With VC^2 users can be aware of uncommitted changes made by remote users; receive request to commit their own changes; be advised to update their local versions. While the final decision is always under user control, the team is made aware of the level of risk when delaying commits and updates in their version control system of choice.

1 Introduction

The development of team projects often requires the use of tools to aid the coordination and synchronization of work between members. There is a subgroup of these tools known as Version Control (or Revision Control) Systems, many of them are widely established and used, specially in software development projects. Popular examples of such systems are CVS [5], SVN (Subversion) [20] and Bazaar [2].

These systems divide in two main subgroups: those using the client-server model (CVS, SVN, ...) and those using the distributed model (Bazaar, Git, SVK, BitKeeper, ...).

In the client-server model, the files of a project are located on a server repository. Each user can checkout her local copy from the repository, work locally in the copy and submit changes back to the server. At any time she can also check the server for new versions of files submitted by other users.

In the distributed model there's no server. Each user has her own local repository with a copy of the project. She may submit changes to her local repository, and at any time, two or more users may choose to merge their local copies.

There are many features available in this kind of systems such as keeping track of changes between different versions of a file, checking the status of a local file against a remote copy, adding and removing files, updating and committing changes, and some that may be specific to the model they use.

1.1 Motivation

While working in a cooperative project under version control tools, users end up concurrently updating the same files. In a client-server system, one of the users will commit first to the central server and the second one will have to incorporate these changes into her files before committing successfully. Version control systems have mechanisms to automatically merge these concurrent changes when different areas of the files are modified. When the same areas of the files are modified, the user has to manually solve the detected conflicts before being able to commit her changes to the central server.

The sole usage of version control tools is not enough to avoid this problem. In typical usage scenarios, these situations occur with some frequency as it was verified in a study with real data in projects hosted at SourceForge.net and supported by CVS [5] (see Section 4 for details). However, solving conflicts may be problematic, specially if the users have made a big number of changes that have not been committed for a long time. These problems may lead users to be reluctant to engage in cooperative projects and even to avoid parallel development [10].

To minimize these problems, users could follow some "best-practices" guidelines, such as regularly keeping their local copies up-to-date and committing as soon as possible, as this approach decreases the probability of conflicts [6]. The problem is that these guidelines are often contrary to good working practices for cooperative projects – e.g. users may only want to commit changes that leave the project in an consistent state, therefore delaying their commits for long periods of time such as days or weeks [17].

The goal of this work is to provide awareness of other users' activities between synchronization points, helping users to coordinate their work and avoid the need for time consuming conflict resolution tasks [7]. Unlike previous works, where awareness is provided in an application that is different from the application used to edit the shared documents [3, 15] or that require specific editors [18, 13], our solution allows users to continue using their preferred unmodified editors. Awareness information is provided as users access files by using any application. Moreover, our solution also does not require any additional infrastructure, building on the existing version control system to propagate the necessary information.

The remainder of this paper is organized as follows: The next section reviews the related work. Section 3 characterizes the semantics of changes in a distributed environment. This is followed by Section 4 that quantifies the amount of conflicts that can be observed in collaborative software development environments. Section 5 presents the proposed architecture and Section 6 illustrates a typi-

cal use case. Implementation details and evaluation are reviewed in Section 7. Discussion and future work in Section 8 is followed by Conclusions.

2 Related Work

Awareness information has long been identified as important for the success of cooperative activities by providing users with an understanding of other users' activities [7]. In groupware systems, a large number of awareness tools have been proposed for synchronous collaboration, such as multi-user scrollbars [1], telepointers and radar views [11], remote screen view [21], etc. These tools allow a user to have some information about the current activities of other users, but they are not appropriate for asynchronous collaboration.

In this work, we address the support for providing awareness information in the context of team projects developed with version control systems. In this context, awareness information can help users coordinate their work, thus minimizing the occurrence of conflicts. For example, if a user starts working on a file, he should be aware if someone else has already modified the file concurrently. Having this information will help users decide when to update files, when to commit, thus minimizing the potential for conflicts.

Version control systems (e.g. CVS [5], SVN [20] and Bazaar [2]) provide the basic support for team projects, by maintaining and controlling the evolution of file versions for the project files. When a user changes a file and commits her changes, it is usually possible to add a comment describing the change. When another user gets the most recent copy of a given file, she can also retrieve the added comments. Although important, this form of awareness is limited as it is only provided when a user decides to update her local copies and it only includes information about committed changes.

CVS also includes an additional mechanism that can be used to provide awareness information: CVS watches. When using CVS watches, users must announce their intents of modifying a file beforehand (by executing a special command). Users can register their interest on specific files and be notified by email when someone announces the intent of modifying it. A user can also check which users have announced the intent of modifying some file. This approach has several limitations. First, using email leads to several undesired properties: users must check for notifications in an application that is different from the one they use for editing the shared files; users may receive notifications while they are working on some unrelated task and they may even forget about the notifications; the delay for email propagation may vary, leading to unpredictable delay for notifications. Second, requiring users to issue special commands for announcing modification intents imposes an non-negligible overhead on users and may lead users to announce intents for all files in the project. These limitations may be partially overcome with integration in a specific editor, such as provided by Eclipse's Team CVS. This tools automatically issues the special CVS commands when the user starts editing a file and it allows a user to check which users are modifying some file. Our work also supports these features, but they can be

used independently of the editor. Additionally, our design is also independent of the version control system and only relies on the common version management functions. Finally, it allows additional interaction among users (e.g. a user may request other user to commit her work).

The BSCW system [3] is a web-based system that includes a version control system to manage shared files. In this system, when some action (check-out, check-in) is executed an event is recorded. The system can present a list of recent event to users when they connect to the system. In [8], the authors introduce a tool for integrating notification and chat with the CVS system. In this system, users are informed when some user commits changes to a file, with events being propagated using an event-dissemination system. Unlike our system, these systems presents no awareness about modifications before they are committed.

In State Treemap [15], the authors propose an awareness widget that allows users to visualize which files are being concurrently modified (leading to a potential conflict) and which locally modified files have already been committed (leading to a conflict). This widget has been integrated in a platform for supporting virtual teams of architects.

The Palantír [18] system provides similar information for files stored in version control systems, relying on an event notification system for propagating information among users. The authors have created wrappers for SVN, RCS and CVS, with events being propagated when edit/update/commit commands are executed. The authors have also developed a plug-in for Eclipse, allowing awareness information to be presented in Eclipse. The Jazz [12] system also provides similar information, as an extension to Eclipse.

In Gasper [13], the authors propose a generic mechanism for propagating limited information about changes being performed. Awareness information is provided in the editors in the form of annotations — e.g. if some user is modifying a method in a code file, other users could see an annotation about this fact in their user interface.

Several other systems have been designed for providing awareness information in the context of collaborative software development (see [19] for a survey). These systems can be divided in two groups. The first (including State Treemap and Palantír) requires the use of an additional tool for checking the awareness information. Besides the problem of convincing users to use an additional tool, this approach has the drawback of requiring users to explicitly check for awareness information when they start editing the shared files (as there is no connection between the editing activity and the tool that provides the awareness information). The second group (including Palantír, Jazz and Gasper) provides the awareness information in the context of a specific editor. This approach is interesting but it forces users to use a specific editor. Additionally, it requires the plug-ins to be updated when a new version of the editor is released. In our work, we provide awareness information for any editor, thus allowing users to continue using their preferred editors. Additionally, unlike previous works that require specific support from the version control systems or that rely on an additional infrastructure, we propagate the required information using files stored

in the version control system. Thus, our approach can be easily deployed with existing version control system.

3 The Semantics of Version Control

In this section we present an overall view of how version control works. At this point we will only consider the centralized model (CVS, SVN, ...) and leave the distributed model as future work. In this presentation, to avoid too much complexity we will not consider the adding and removing of files and folders from the repository and the conflicts that may arise from these operations.

In version control systems, a user must start by checking out files that are stored in the repository, thus creating a local private copy. At any moment, the user may update her local copy against the latest version stored in the repository **(up)**. The user may also modify her local copy **(m)**. After modifying her files, the user may commit her changes to the repository **(c)**. When considering other users activity, two additional actions may occur on a given file. A remote user may have modified the file **(rm)** or committed her remote changes **(rc)**.

From the point of view of the local user, the state of a file may combine the following situations:

- Remotely changed **(R)**, when a remote user has modified her local copy of the file but she has not committed her changes yet.
- Outdated **(O)**, when a remote user has committed changes to a file that have not been incorporated in the local copy.
- Modified **(M)**, when the file has been locally modified.

In Figure 1 we present a diagram that shows the possible states of a file, and the transitions among these states induced by users' actions (both local and remote actions). There are two situations that may lead to a conflict if the user modifies her local copy:

- The user modifies a file that was remotely modified by some user that has not yet committed her changes to the repository.
- The user modifies a file that is outdated against the current version on the repository.

Conversely, conflicts may also arise if a remote user modifies an old version of a file that has been locally modified, either it has already been committed or not. In the figure, we mark as **thin-dashed** all these dangerous transitions that may lead to a conflict status. For each of these transitions, we mark as **thick-dashed** the alternative transition that should be taken to avoid falling into a conflict situation. All other transitions are considered normal actions of working in the file.

In order to help the user avoiding the situations that may lead to conflicts, our tool will provide advance warning about conflict-leading actions. Thus, when a users starts accessing a file that has been modified elsewhere, a notification

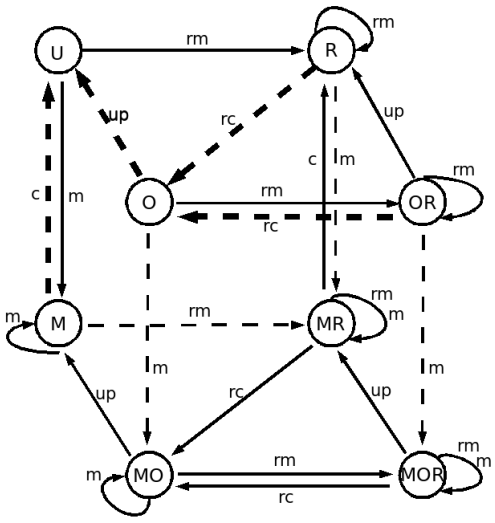

Fig. 1. Transitions between states of a file from a local perspective.

will be presented to the user suggesting an alternative action. For example, if the user accesses a file that is being concurrently modified by some other user, the user is notified of the fact and may ask the other user to update the file.

4 Conflicts on Real Usage Scenarios

To assess the importance of providing an awareness tool for version control systems, we have decided to investigate the degree of conflict occurrence in real usage scenarios. In the distributed filesystem community, several usage studies have been presented showing that, although supported by these systems, there is minimal file sharing and that concurrent updates are very rare (e.g. in [16], the authors report 0.025% for the update conflict rate and 0.004% for the unsolved conflicts).

It is clear that in cooperative settings the situation should be different and a much higher conflict rate is expected. To confirm and measure this, we have decided to study real traces from collaborative software projects managed by CVS at SourceForge.net. In this paper we only present a brief summary of our results (the complete results are presented elsewhere [4]).

In CVS, for each project, the server maintains a log with all accesses to the files of the project. For studying conflicts, we have downloaded the logs for several multi-user projects and we have computed the following statistics:

number of updates executed by users[3]; number of conflicts (either automatically merged or requiring manual resolution); and the number of conflicts requiring manual resolution. We have selected projects in the most *active list* (for which we have analyzed a period of two months of activity) and other less active projects (for which we could analyze the full log).

These logs had to be processed before statistics could be obtained. First, after solving a conflict manually, a user must commit the new version, leading to two CVS log entries. Second, we have observed that, after a conflict detection, users sometimes download the current version of the file before committing a new version. This suggests that users prefer to re-apply their changes to the new version instead of changing the version produced by CVS. In both cases, we count the resulting pair of log entries as one unsolved conflict.

Table 1. Statistics of conflicts (concurrent updates) in several collaborative software development projects hosted at SourceFogre.net.

				Conflicts in		
Project	# users	# files	# updates	top 10%	top 50%	all files
Gallery	24	2347	4456	17.1%	6.3%	3.1%
Gnuplot	13	795	15813	14.8%	5.2%	2.6%
Xine	32	1460	2359	11.7%	4.6%	2.3%
Firebird	91	7592	50370	9.4%	2.0%	1.0%

In Table 1, we summarize the most relevant information about the logs processed, including the number of users and files involved in the development of the project and the number of updates executed during the observed period. We also present the rate of conflicts when considering the 10% most modified files, the 50% most modified files and all files. In Figure 2, we detail these results for the Gnuplot Project. For ease of presentation, each point in the graph presents the average of 20 files.

The results show that the rate of conflicts is large. As we could expect, when considering only the most active files, the rate of conflicts is much larger as it is more likely to have more than one user modifying the file. For rarely accessed files, conflicts are rare. For example, when considering only the 10% most active files, the rate of conflicts ranges from 9% to 17%. From the results of the Gnuplot project, we can also observe that almost half of these conflicts have to be solved manually by the users. Thus, it seems clear that the existence of a tool for providing awareness information about other users' activity may improve the effectiveness of the collaborative process by helping users to avoid these conflicts.

[3] Using the CVS logs, an update can only be detected when the user decides to commit her changes. Thus, one update may reflect several changes performed by a single user from the last moment she has updated her local copy of the file until she decides to commit her changes.

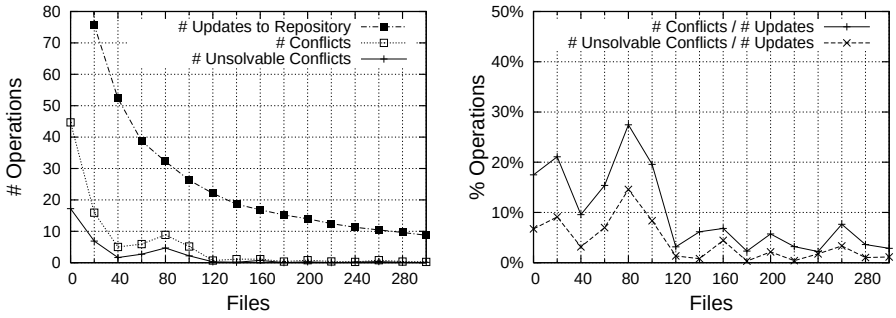

Fig. 2. Gnuplot Project statistics - absolute values on the left graph and the ratio between solved/unsolved conflicts against the number of updates on the right.

5 System Architecture

5.1 Approach

The proposed Version Control Control tool (VC^2) provides awareness for cooperative editing activity by combining filesystem probing and activity dissemination among the group using metafiles stored in the version control system.

A first step is to detect activity on any file under version control. This is achieved by inserting probes in the filesystem of participating nodes. Striving for a general solution, we avoided considering particular file activity patterns of specific editors, but instead concentrated on the detection of distinctive actions that result from the user activity. When a user starts editing a file, the editor application has to read the file from disk. We detect this action by detecting the *open* system call. When a user decides to save her changes to the file (or when the application automatically saves her changes), the editor application has to write the file contents. We detect this action by detecting the *close* system call after the new file version being written to the disk.

Filesystem probing can be achieved by adding code to kernel filesystem routines, with kernel tapping mechanisms such as FUSE [22] and FIST [24], or by subscribing to filesystem events on kernels that support them, e.g. INOTIFY[23]. Since VC^2 notifications do not require blocking or delaying file operations, kernel access is not mandatory and event subscription can suffice. This also means that users are warned of actions that can lead to conflicts but are free to ignore those warnings and have full accesses to the files.

A second aspect concerns the diffusion of information within the developer's group. While other systems choose to use external mechanisms of dissemination or tailor their own dissemination service, in VC^2 we choose not to add any additional mechanism. This is possible because an existing communication mechanism is already available in the version control mechanism.

VC^2 uses the existing version control system to disseminate additional control information across the hosts. It suffices to enclose this information as spe-

cial metafiles in the project tree and exchange them with the existing commit and update functionality. This also allows reuse of authenticated channels, say SSH, that might already be prepared for communication with the project server. Finally, this approach provides awareness information even if users are not connected at the same time, because the control information is maintained on the server. For example, if a user has started to change her local copy of the file but then she has stopped working and disconnected her computer, the information about this activity is still stored in the server. If another user (that could have been disconnected) starts accessing the file, he will be notified of the potential problem.

5.2 Architecture

The typical setup, when using the tool with client-server version control, comprises a CVS/SVN server in a hosting server machine and two or more users in a given number of other machines – we will refer to these as clients. Client machines must have connectivity and access to the server, either by SSH or specific CVS/SVN ports. There is no need for connectivity among client machines. There is no impact even if they are all hidden by firewalls.

Any client machine will just have to support standard CVS/SVN client tools and, in addition, have installed local support for either FUSE or INOTIFY (See Section 7 for supported systems). The VC^2 approach tolerates the presence of standard CVS/SVN clients that cannot introspect the filesystem and run the tool. As expected, activity on those clients will not be made aware to others and vice-versa.

Each client machine runs a VC^2 daemon that acts on filesystems events, creating separate threads for any event that requires GUI interaction or communication to the server. Periodically, the daemon pulls information from the server in order to detect reported remote changes on monitored files. In the following section we give details on the behavior of VC^2.

5.3 VC^2

The tool currently supports the usage of CVS and SVN (since these are the most popular open source tools using the client-server model), although support for other systems could be easily added.

The daemon is implemented as a Java user level process. It makes use of a filesystem notification layer that can be provided by either FUSE probes or INOTIFY events (details on these tools will be given in Section 7). Once aware of all system calls to the filesystem, our tool can check if the accessed files are under control of any version control system, and try to detect behaviors that may potentially lead to conflict scenarios. If such a case is detected it will alert the user suggesting a recommended action to take.

To start using our tool it suffices to have a working setup of the supported version control systems and start the Java VC^2 daemon. At this point the user can checkout her projects to directories inside a controlled local filesystem. After

that, every time she opens or closes a file, the system will check if that file is under control of the version control system. All other files are ignored and are transparent to VC^2.

If a file is under control, our tool will check its status and alert the user when necessary. The status of the file is kept in a metafile which is saved in the repository. The first time the tool checks the status of a file, it create the associated metafile if none exists. Each time the system consults or changes a metafile, it will update it from the repository and if there are changes it will commit immediately. Inside the daemon, we implement concurrency control to guarantee the atomicity and consistency of these operations. All metafile operations on the repository are made with standard CVS/SVN tools.

At the moment, the only information on the metafile is the number of users with uncommitted changes, and the number of users requesting the commit of those changes. In Section 8 , we will elaborate on additional information that could be used in the metafiles, such as the name of the users changing/requesting a file and messages associated with these changes/requests.

When opening a file, our tool will consult the metafile to check for uncommitted changes made by other users. If there are any, it will alert the user of the situation and ask if she wants to request the updating of that file. Also, if the local copy of the file is outdated compared to the version on the repository, it will alert the user and ask if she wants to update to that version.

When closing a file, it will check if there were any changes made, by checking the status against the repository version. If the file was locally changed it will increment the number of changers on the metafile. The file will also be added to a list of uncommitted files, that is regularly checked by a timer thread.

The timer thread checks the list of uncommitted files by two reasons. The first is to check if any user has requested the update of a locally modified (but uncommitted) file. If one is found, it alerts the user about this request and asks if she wants to commit her changes. The other reason is to check if the user already manually committed the files and they are no longer locally changed. In any of the previous cases, if the file was committed, the number of changers is decremented, and if it reaches zero (which means that there is no uncommitted changes) it sets the number of update requesters to zero.

User alerts are implemented by dialog boxes that pop up on the screen, usually with a yes/no question. Since we are using Java, our tool is cross-platform. There is a small delay between the open/close of the file and the alert. This is the time spent to update the associated metafile, and will depend on the latency of the connection to the repository. It may be almost imperceptible if the repository is in a local network or the connectivity is fast, or it may take up to a few seconds if the user has bad connectivity to the repository server.

As it was described, there is no need for server side intervention on CVS/SVN platforms, allowing the use of any off-the-shelf public server. Additionally, as it relies only on common version control management, it should be immediate to include support for other version control system.

6 Usage Example

In this section we show a small example of how the tool is used and how it helps the coordination of a team project.

Suppose there are three developers working on an AddressBook application in C++. Alice is developing the user interface, Bob is working mostly on the core of the application, and Charlie is developing a storage module to save application data. The structure of the project is the following:

```
AddressBook/inc/core.h
AddressBook/inc/storage.h
AddressBook/inc/UI.h
AddressBook/src/core.cpp
AddressBook/src/storage.cpp
AddressBook/src/UI.cpp
AddressBook/help.html
```

The developers start by checking out the project from the CVS repository into a local directory, and then begin working on it. Suppose the following action are executed:

1. Bob changes *core.h* and *core.cpp*
2. Alice which has been working on *UI.cpp* now needs to know how to retrieve phone numbers and consults *core.h*. As she opens the file, a popup alerts her that there is one user with uncommitted changes on that file, asking if she wants to send an update request (Figure 3). She agrees.
3. A popup appears on Bob's screen saying that one user is requesting an update on *core.h*, and asking if he wants to commit (Figure 4). Bob agrees and the file is committed.
4. Alice, who decided to work on something else, now returns to her previous task. As she opens *core.h*, a popup alerts her that her version of the file is outdated, asking if she wants to update. She agrees and the file is now up-to-date.
5. Charlie, who's been working on the storage has committed a new version of *storage.cpp* and is already working on a new one.
6. After running some tests, Bob believes there may be a bug on the storage, and decides to consult *storage.cpp*. He is alerted to the fact that there is a user with uncommitted changes on that file, and agrees to request an update. He is also alerted to the fact that his version of the file is outdated, but decides not to update immediately and waits until Charlie commits his recent changes.
7. Charlie is alerted about one user requesting him to commit his changes on *storage.cpp*, but as he is currently fixing a bug, he decides not to commit until he finishes.
8. A few minutes later the bug is fixed and Charlie commits his changes. In the mean time, both Alice and Bob are making some changes to the *help.html* file.

9. Charlie wants to add this bug-fix in the release notes of the *help.html* file. He is alerted to the fact that two users have uncommitted changes (Figure 5). Since he just wants to add a line of text, he decides not to request an update.
10. Later, Bob decides to check again for the bug. He updates the storage files, runs some new tests and realizes that the bug has been fixed.

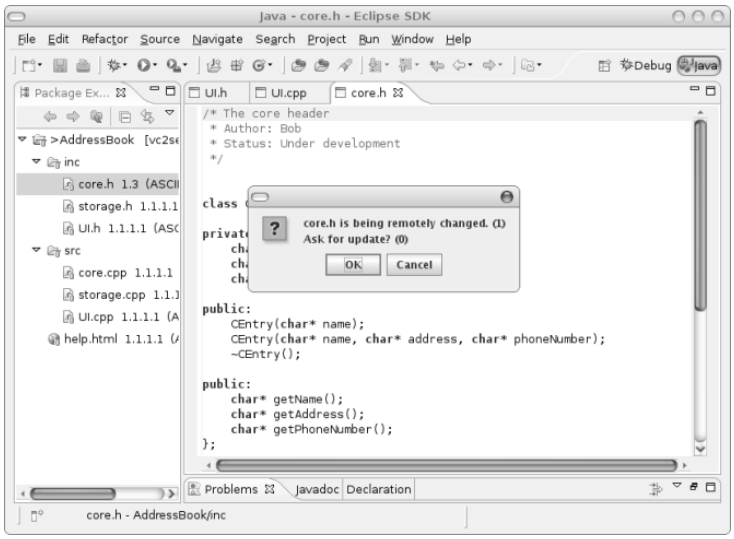

Fig. 3. Alice being alerted that the file she wants to access has been modified by some other user. In this case, Alice is using the Eclipse editor.

This is a little example of how awareness can help the coordination of software development. Its simplicity may not fully expose the importance that a small increase in awareness can have in team development, but this becomes clear when we consider the number of conflicts that occur in real situations, as reported in Section 4.

7 Implementation

We have created two implementations of the filesystem layer of our tool using different approaches. The first is based on FUSE [22] and the second one on INOTIFY [23]. Each has its own advantages and disadvantages, that are discussed throughout this section.

7.1 FUSE

File System in Userspace (FUSE) is a Unix kernel module that allows non-privileged users to create virtual filesystems that run in user space [22]. The

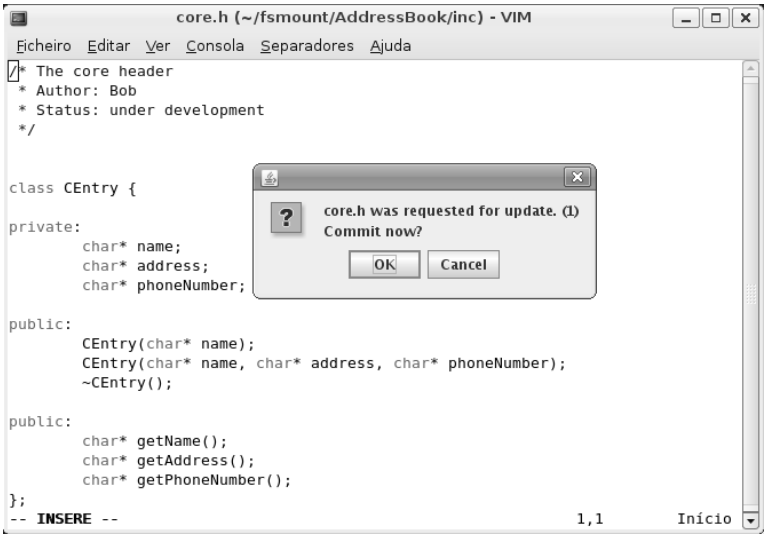

Fig. 4. Bob being asked to update his uncommitted version of the file (as a result of Alice's request). In this case, Bob is using the VIM editor.

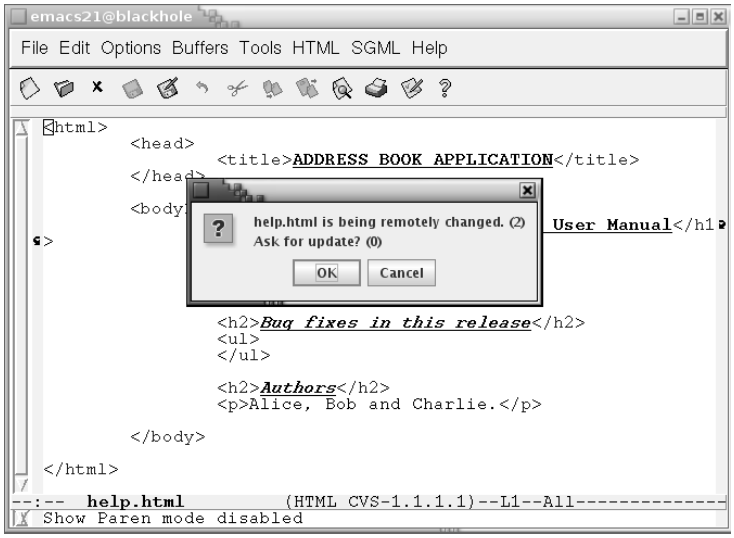

Fig. 5. Charlie being alerted that the file he want to access has been modified by some other user. In this case, Charlie is using the Emacs editor.

FUSE module intercepts system calls to the filesystem and redirects them to code that runs at user level. There are many projects using FUSE to create a virtual filesystem with different purposes. Some popular examples are [22]:

- SSHFS: Provides access to a remote filesystem through SSH
- GmailFS: filesystem which stores data as mail in Gmail
- EncFS: Encrypted virtual filesystem
- Captive NTFS, ntfsmount, and NTFS-3G, allowing access to NTFS filesystems

FUSE is available for Linux, Mac OS X, Solaris and FreeBSD. It is also possible to implement similar mechanisms in Windows (which are currently used, for example, by anti-virus and indexing software). There are several language bindings available such as C++, Java, C#, Haskell, Python, Perl and others.

In this project we used FUSE-J [9], the FUSE binding for Java. This solution has the advantage of intercepting the filesystem calls, allowing to execute code before returning results to the applications. This could be used, for example, to update the local copy of a file before allowing the application to read its contents. These kind of functionality is not used in our tool, as we have decided to just provide awareness information.

7.2 Inotify

INOTIFY [23] is a Linux kernel subsystem that provides filesystem event notification (a similar mechanism exists in the Mac OS X system). With INOTIFY it is possible to monitor directories and files for events such as *open*, *close*, *create* or *delete*. An application may register itself to be notified for events occurring inside a set of directories, which is much more efficient than actively searching for changes or interposing code in the execution if the file system calls (as in FUSE).

In this project we used JNOTIFY [14], a Java library that works as a wrapper around the INOTIFY API.

7.3 Benchmarking

In this section we do some performance analysis on our tool. We want to see how our virtual filesystem compares to the native filesystem. For this, we used a project with 100 text files with a total of 1.7 MB (average file size: 17 KB).

As described earlier, each time the user opens a file, our tool will check if it is under version control, and in this the case it will check if there are remotely uncommitted changes or if the file outdated. This check is asynchronous, which means the user can keep opening files while another thread performs the check. We measured the time spent to open and read all the files (using the command: `cat * >/dev/null`) in the following scenarios: on the native filesystem; FUSE *vs* INOTIFY; normal files *vs* files under CVS control *vs* files under SVN control. In all cases the repository is on a remote machine. On Table 2, we present the

Table 2. Benchmarking results (in milliseconds per file) for the 1st run, the mean of 10 runs and standard deviation.

	Native	Inotify	Inotify+CVS	Inotify+SVN	Fuse	Fuse+CVS	Fuse+SVN
x_1	0.206	0.208	0.209	0.399	11.913	24.906	33.179
$\overline{x}$	0.185	0.220	0.299	0.383	11.569	13.898	14.174
σ	0.014	0.054	0.285	0.557	0.203	3.688	6.502

results for the first run, followed by the mean value of 10 runs and the standard deviation.

From the results in Table 2 we can observe that the overhead of using INOTIFY is almost null. Since notifications do not delay the filesystem call flow, the overhead is only due to the increased load on the machine and the running of additional tasks. The system asynchronously verifies if some awareness information must be provided. The delay to provide such information depends on the latency to the repository server, as the client's daemon must check if there is any recent control information on the server.

It was not unexpected to confirm that there is a significant time overhead associated to filesystem interception in FUSE. In particular, on the first access to each file, when the metafiles do not exist and have to be created. It could be argued that the loss of performance caused by the virtual filesystem does not affect the normal development of a project, since the overhead in opening a single file is not perceived by the user in an interactive session (the incurred delay is below 15ms). However, there are still situations where it can have a greater impact. The initial checkout of a large project will be considerably slower on the virtual filesystem.

In the current VC^2 implementation there is no situation in which we opted to delay or deny user actions on files. There are cases in which one could want to do so, for instance if we wanted users to have mandatory interactions with the VC^2 GUI in order to proceed with the opening of files. In such cases, filesystem interception would be the only solution.

Under the present VC^2 interaction model the best solution is clearly to resort to filesystem notifications and avoid the overhead incurred by file system interception.

8 Discussion and Future Work

The simple and generic architecture of our tool makes it easy to extend it with new features. As we mentioned before, there are still many features worth of development. In this section we discuss some of these features.

Knowing the number of users that are changing/requesting a file is useful, but this can be further improved by showing a list with these users' names. To this end we may consult the username used to access the repository, or the username on the local machine. Along with the list of users changing/requesting a file, we could also associate messages with these actions by having a small text

input field in the popup dialogs. These would be stored on the metafiles and displayed in the alert messages. This can provide a lightweight communication channel with a history that may be integrated in the version control system's logs (as in [8]).

When a user is alerted to the fact that other users have uncommitted changes and sends a request for update, it would be useful to be alerted when all the users have committed. The timer thread could be easily extended to verify this situation (as it already verifies a similar situation on locally changed files).

As discussed before, it may not be convenient for developers to immediately commit their source code. This is not the case of documentation and other kind of files. For these we could define user configured properties such as *auto-commit* or *auto-update* that would automatically commit a changed file, or automatically update outdated files, and consequently eliminate most of the need for user interaction.

8.1 User Interface

One of the main goals of our tool is automatic integration with minimum user interaction required. The users do not need to explicit consult the state of the project's files to receive awareness information. Instead, they work normally and eventually receive alerts for potentially dangerous situations. Still, our alert-based approach is a bit intrusive, creating a new window for each alert. We are studying the possibility of providing an alternative interface, where all alerts could be cumulatively displayed in a single, less intrusive, popup window.

Although some users may enjoy the simplicity of using just our alert based solution, others may require additional functionality such as checking the activities of other users in the context of some project (as provided, for example, by [15, 18]). In our tool, this information is already maintained in the metafiles. For supporting these users, we are currently developing an user interface that allows the user to browse this information (e.g. check which files are being modified, and by whom, who has requested updates and other info we may extend). In this tool, we could also allow users to apply an explicit action such as updating, committing, requesting update, or even set auto-update/commit flags.

9 Conclusions

Conflicts are bound to occur when users engage in collaborative development of code (or text) in standard revision control frameworks. Since conflicts are often troublesome to solve, users try to avoid them by frequently issuing commits or by negotiating in parallel channels, like instant messaging applications and email. These actions are preventive and time consuming, and they are necessary due to the lack of mutual awareness among developers.

In this paper we introduce a system that addresses this problem, by greatly improving the level of awareness without forcing the commitment of unstable versions. The solution depicted in VC^2 is general and independent from the

actual version control system in use. It currently runs on CVS and SVN and it is straightforward to adapt it to other version control tools. More important, VC^2 does not require any intervention on the server side, thus the solution can be deployed with public servers, such as SourceForge. Our solution also works independently of the editor used by users, thus allowing users to continue using their preferred applications.

References

1. Ronald M. Baecker, Dimitrios Nastos, Ilona R. Posner, and Kelly L. Mawby. The user-centred iterative design of collaborative writing software. pages 775–782, 1995.
2. Bazaar. Bazaar version control, 2007. http://bazaar-vcs.org/.
3. R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel. Basic support for cooperative work on the world wide web. *International Journal of Human Computer Studies: Special issue on Novel Applications of the WWW*, 46(6):827–856, Spring 1997.
4. Marcos Bento. Desenho e implementação de um sistema de ficheiros com suporte para dispositivos de armazenamento portáteis. Msc thesis, FCT - Universidade Nova de Lisboa, 1 2007.
5. Per Cederqvist et al. Version management with CVS, 2007. http://www.cvshome.org/docs/manual.
6. S. Dekeyser and R. Watson. Extending Google Docs to Collaborate on Research Papers. Technical report, The University of Southern Queensland, Australia, 2006.
7. Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114. ACM Press, 1992.
8. G. Fitzpatrick, P. Marshall, and A. Phillips. CVS integration with notification and chat: lightweight software team collaboration. *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 49–58, 2006.
9. FUSE-J. The fuse binding for java, 2007. http://sourceforge.net/projects/fuse-j.
10. R.E. Grinter. Using a configuration management tool to coordinate software development. *Proceedings of conference on Organizational computing systems*, pages 168–177, 1995.
11. Carl Gutwin, Mark Roseman, and Saul Greenberg. A usability study of awareness widgets in a shared workspace groupware system. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 258–267, New York, NY, USA, 1996. ACM Press.
12. Susanne Hupfer, Li-Te Cheng, Steven Ross, and John Patterson. Introducing collaboration into an application development environment. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 21–24, New York, NY, USA, 2004. ACM Press.
13. Claudia-Lavinia Ignat, Gérald Oster, Pascal Molli, and Hala Skaf-Molli. Gasper: A collaborative writing mode for avoiding blind modifications. Research Report RR-6204, LORIA – INRIA Lorraine, may 2007.
14. JNotify. Jnotify linux api, 2007. http://jnotify.sourceforge.net/.
15. Pascal Molli, Hala Skaf-Molli, and Christophe Bouthier. State treemap: an awareness widget for multi-synchronous groupware. In *7th International Workshop on Groupware - CRIWG'2001*, Darmstadt, Germany, September 2001.

16. T. W. Page, Jr., R. G.. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), 1997.
17. R. Robbes and M. Lanza. Versioning systems for evolution research. *Proceedings of IWPSE*, pages 155–164, 2005.
18. Anita Sarma, Zahra Noroozi, and Andre van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.
19. M.A.D. Storey, D. Čubranić, and D.M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. *Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202, 2005.
20. Subversion. Next-generation open source version control, 2007. http://subversion.tigris.org/.
21. Kimberly Tee, Saul Greenberg, and Carl Gutwin. Providing artifact awareness to a distributed group through screen sharing. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 99–108, New York, NY, USA, 2006. ACM Press.
22. Wikipedia. Filesystem in userspace — wikipedia, the free encyclopedia, 2007. [Online; accessed 30-May-2007].
23. Wikipedia. Inotify — wikipedia, the free encyclopedia, 2007. [Online; accessed 6-June-2007].
24. Erez Zadok and Jason Nieh. Fist: a language for stackable file systems. In *Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2000. USENIX Association.