Forby: Providing Groupware Features Relying on Distributed File System Event Dissemination*

Pedro Sousa[1], Nuno Preguiça[1], Carlos Baquero[2]

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

Abstract. Intensive research and development has been conducted in the design and creation of groupware systems for distributed users. While for some activities, these groupware tools are widely used, for other activities the impact in the groupware community has been smaller and can be improved. One reason for this fact is that the mostly common used applications do not support collaborative features and users are reluctant to change to a different application. In this paper we discuss how available file system mechanisms can help to address this problem. In this context, we present Forby, a system that allows to provide groupware features to distributed users by combining filesystem monitoring and distributed event dissemination. To demonstrate our solution, we present three systems that rely on Forby for providing groupware features to users running unmodified applications.

1 Introduction

Intensive research and development has been conducted in the design and creation of groupware systems. While for some activities, these groupware tools are widely used (e.g. instant-messaging, conferencing tools), for other activities the impact of the groupware community can be further improved. One example is the support of asynchronous cooperative editing, where version control systems (e.g. CVS, subversion) are widely used for data management but the proposed awareness mechanisms [5, 17, 14] are rarely used in practice.

One reason for this fact is that the mostly common used applications do not support collaborative features and users are reluctant to change to a different application. This fact has been pointed out before [18] and several approaches for providing groupware features in existing applications have been proposed in literature, as reviewed in the related work section.

In this paper we propose a different approach, relying on the combination of modern file system mechanisms and an event dissemination system. Current operating systems include lightweight file system mechanisms to monitor files accessed and modified by users, allowing to infer user activity based on monitored actions. By combining the information obtained from these mechanisms with

* This work was partially supported by FCT/MCES (POSC/FEDER #59064/2004 and PTDC/EIA/59064/2006.)

an event dissemination system to disseminate this information among multiple users, it is possible to provide groupware features to users in a distributed environment. As exemplified in the applications presented in section 5, a wide range of functionalities can be provided, from awareness information to new groupware applications.

As far as we know, this approach has been previously neglected by the groupware community. However, when compared with other proposals, it has the advantage of being application-independent, allowing the developed solution to work with all applications, thus providing groupware features while users can continue using their preferred applications. A disadvantage of this approach is that it can only be used when file system activity occurs. This limits the scenarios where our approach can be used. Thus, we think that this approach is not intended to be a replacement of existing proposals, but rather a complement to such proposals.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents existing file system mechanisms and surveys their availability in the most popular operating systems. Section 4 presents Forby design. Section 5 discusses how to use Forby in the context of three different applications. Section 6 concludes the paper with some final remarks.

2 Related work

Several approaches have been proposed to integrate or improve groupware features in existing applications. In this section, we survey some of these proposals, grouping them by their approach.

The first approach is to provide application sharing, by allowing a group of users to transparently share the same application. Systems like Microsoft NetMeeting provide such functionality. However, as discussed in [2], this approach has a series of shortcomings, such as the lack of support for real concurrent work and large network bandwidth requirements.

The second approach consists in dynamically replacing single-user interface components of applications by multi-user versions. The replacement occurs at run-time and is transparent to the original application. Flexible JAMM [2] provides this functionality. This approach is very effective, but its applicability tends to be limited in the number of applications that can be supported.

The third approach consists in creating minimal modifications to existing applications or relying on information already provided by applications. For example, in [6] the authors present a generic mechanism for awareness applied to the BSCW shared workspace. In this context, events that are to be propagated are obtained by sensors that are integrated with the BSCW server. In [12], the authors use existing operations for getting and setting the state to provide synchronous editing functionalities in existing editors.

This approach is interesting but solutions are application specific. Thus, for providing the same functionality in different applications, the solution (or part of it) must be re-written, thus limiting its applicability. Additionally, when new

versions of the software are released, it is necessary to verify if the developed solutions still work with the new version.

The fourth approach is based on the use of application plug-ins. In QnA [1], the authors present a plug-in for commercial instant-messaging clients that increase the salience of incoming messages that may deserve immediate attention. For the open-source Eclipse editor, plug-ins developed in Palantír [16], Jazz [7] provide different levels of awareness information for shared files.

This approach has the same shortcoming of the previous approach, with the difference that plug-ins are officially supported and it is expected that new versions do not break existing plug-ins.

Some applications do not directly support plug-ins, but some extension can still be achieved (e.g. by intercepting application events at the operating system level, such as mouse movements, keyboard activity, etc.) This approach has been used in several systems [18, 11] for integrating groupware features in applications such as Microsoft Word. This approach has the same problems as the previous ones, but it is even worse as there is no supported API for the created extension.

In the Placeless Documents system [4], it is possible to associate with each document some code, in the form of active properties, that is run when the document is accessed. This approach can also allow to provide groupware features to existing applications. However, unlike our approach, for this code to be executed, the files must be accessed through the Placeless Documents interfaces, that includes a distributed file system interface. Additionally, unlike the solution proposed in our system, this system provides no support for combining applications running in different sites, thus making it harder to provide groupware features in distributed settings.

3 File system mechanisms

File system support has been evolving, with the introduction of new functionalities. Although different operating systems provide different mechanisms, most modern operating systems tend to provide the same file system functionalities relying on different APIs. In this section we overview some of the new features that can be used when providing groupware features.

3.1 Notification mechanisms

Notification mechanisms allow user level applications to be notified when some access to the file system is executed. This includes the execution of open, read, write and close operations on files and changes in directories.

Notification is performed asynchronously, out of the loop of the system call, thus imposing minimal overhead to the operation execution. This approach also alleviates the efficiency requirements for the functions that process notifications, as the execution time is not reflected in the original operation execution.

These mechanisms are currently used in several applications, such as indexing systems for tracking changes in files and directories without requiring expensive periodically polling of the file system.

Notification mechanisms are available in the three most popular desktop operating systems. In Windows, Win32 API supports this feature natively for some time. Additionally, the .NET framework also supports this feature using the *FileSystemWatcher* library class, making it available to all supported languages. In Mac OS, the File System Events API provides such functionality since Mac OS v. 10.5. In Linux, the Inotify subsystem provides such functionality and it is integrated in the kernel since version 2.6.13. Other subsystems provide similar functionality (e.g. FAM, dnotify).

These APIs are available to user level applications, making the implementation and debugging of applications that use these mechanisms simple. Additionally, for each of the proposed mechanisms, there are mappings to several programming languages allowing to access these features easily in different programming environments. In the Java language, the JNotify library provides similar support in both Windows and Linux systems.

The APIs for all these systems consists in a very reduced number of methods, that can be used in a very straightforward way. The example in Figure 1 shows how to use Inotify for requesting notification of accesses to a file. It also shows how the application can have access to the notifications.

```
//***** Initialize Inotify ************************
int fd = inotify_init ();
if (fd < 0) return -errno;
//***** Add a Watch *******************************
int wd = inotify_add_watch(fd,
    "/home/user/myfile", IN_ACCESS | IN_MODIFY);
if (wd < 0) return -errno;
//***** Read an inotify event *********************
static struct inotify_event buffer;
*nr = read (fd, &buffer, sizeof(buffer));
```

Fig. 1. Setting up a notification request for accesses on a file and obtaining notifications in Inotify.

3.2 Interception mechanisms

Interception mechanisms allow to intercept file system calls and execute code before returning the result. This code may execute any action the system developer wants, including executing direct access to the file systems, sending messages over the network or even denying file operations by returning an error. This mechanism is used, for example, by anti-virus software to avoid that users run compromised software.

Unlike the previous mechanism, file system interception is executed in the path of the system call. Thus, the time consumed in executing the additional code is directly reflected in the delay the application experiences when executing the file system operation. Thus, the implemented code must be efficient, especially

when no additional functionality is being executed, or otherwise the system performance will be affected.

The existing interception mechanisms can be broadly divided in two classes: kernel-based and user-level based. The first, run fully inside the kernel of the operating system. This approach is more efficient, as no additional user/kernel transition is necessary. However, it is much more complex to develop applications using this approach, as debugging code that runs in the kernel is very difficult (an error may lead to a freeze in the system). The second class, only includes a small module running in the kernel that intercepts the file system calls and redirects them to user level, where the additional code runs. This is less efficient than the previous solution, but it is much simpler to develop. Additionally, it is always possible to use this approach during the development phase and migrate the developed code to the kernel after the development is completed. Concerning user level solutions, it is also possible to intercept file system calls executed using dynamic linked libraries (e.g. the C library in Unix-like systems) by replacing the code of these libraries. This approach is efficient but it is not generic, as it only works for applications that use dynamic libraries for executing file system calls.

Several approaches exist to implement file system call interception. *File System in Userspace* (Fuse) is available for both Linux and Mac OS operating systems. This system uses a small kernel module that redirects intercepted calls to user level. Application developed using Fuse run in user level and must specify, for each file system call, the code to be executed. Fuse is very popular and simple to use, and a large number of systems were developed on top of it. For Linux (FreeBSD and Solaris), FiST [19] provides a kernel level solution that includes a specific language for simplifying the creation of stackable file systems. In the Windows operating system, the *Installable File System* API allows to intercept file system calls at the kernel level. An application implemented using this API can either fully execute in the kernel or redirect the call to user level.

4 Forby

Forby is a middleware system for providing groupware features to distributed users. The system provides two main components, present on each node: a file system monitoring module and a distributed event dissemination module, as shown in Figure 2. The file system monitoring module is used to infer user actions, based on file system activity. The distributed event dissemination module is used to propagate information among nodes. Applications use these two low-level components for providing groupware features to users. Next, we detail the two main components of Forby and later discuss how to build an application relying on these components.

4.1 Monitoring component

The file system monitoring module monitors the user's working area, capturing the actions made by that user, processing them and forwarding them to the

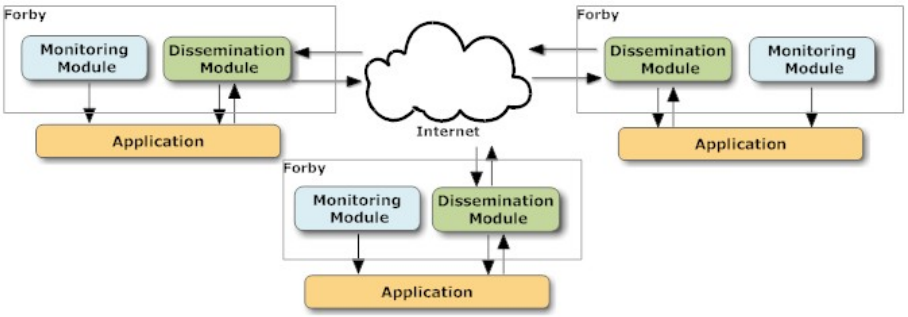

Fig. 2. Forby architecture.

application if those events are important. This component is divided in three sub-modules that perform the operations required to capture, pre-process and deliver the important file system events to the application. The sub-modules are the activity monitor, event filter and event manager (Figure 3).

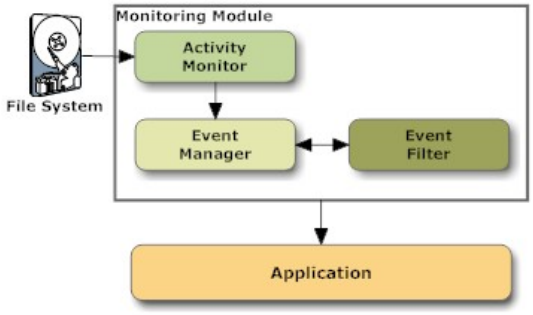

Fig. 3. Forby monitoring module.

The activity monitor sub-module is responsible for monitoring the activity in the file system, obtaining the events through any existing tool or file system mechanism. Whenever a user performs an action in the monitored area, an event (*FSEvent*) is created. The FSEvent includes the information about the file, the access type and the current system time. The system can monitor any type of file access – create/delete, open file, read, write, etc.

This is the only module that is operating system dependent. Currently, there are two implementations based on *JNotify* [10] running on Windows and Linux systems and one implementation based on *Installable File Systems* (IFS) running on Windows systems. Forby is prepared to easily integrate other file system monitoring mechanisms by only creating a new activity monitor sub-module.

When using Forby, one activity monitor sub-module must be selected in each node, but different nodes of the same application may use different activity monitors.

As file system monitoring generates a large number of events, to simplify application development it is important to filter the messages propagated to the applications. Additionally, applications are often more interested in knowing that a given pattern of accesses have been executed, rather than having to process the all event sequence. For example, an application that just needs to know when a file has been changed, is not interested in processing the complete sequence of open, reads, writes, close events but rather prefers to receive a single event notifying that the file has been changed. The event filter sub-module is responsible to achieve this functionality by pre-processing the raw events.

As the necessary events are application-dependent, the application must define the event filter to use. Although the programmer can define a new event filter from scratch, a set of filters with basic functionalities is available for re-use. The basic functionality of these filters is to keep information about which files or directories should be monitored or ignored, updating the list of files/directories as the result of create/delete operations. All events on files that must be ignored are immediately discarded.

On top of this file management functionality, two basic filters were defined. The first, simply filters the events based on their type using some defined mask without further processing. The second, groups sequences of events into more complex events – for example, a sequence of write (resp. read) operations is notified as a file modification (resp. file access) event. These basic filters can either be used as is by the applications or be further extended. Our experience in developing applications using Forby shows that the basic filters can usually be used without changes, as application requirements tend to be similar.

The use of event filters makes sure that the application only receives the important events, therefore reducing the number of events that are processed. This approach simplifies the application's design and coding, since most of the effort of filtering the events is made at the event filter sub-module. Additionally, this is also important as it contributes for reducing the processing needs by filtering events as soon as possible, thus minimizing the overhead of Forby and improving system performance.

Finally, the event manager manages the events, linking the monitoring system to the application. It receives the events from the activity monitor, validates and pre-processes them using the event filter and ultimately, if that is the case, forwards the events to the application.

4.2 Dissemination component

Although any event dissemination system could be used for this purpose, Forby dissemination component implements a system specially tailored for supporting users with minimal infrastructure requirements, thus allowing users to easily use the system. To this end, the event dissemination system manages the group

participants, organizing them in a peer-to-peer dissemination tree that allows efficient event exchange among the peers.

Event dissemination is epidemic, where each participant that receives an event must forward the event through the tree, so that the event reaches all participants. Also, when an event is received, it is delivered to the system application in order to be processed.

For joining the dissemination tree, a new group member must obtain a list of participants already in the group, by contacting a *rendez vous point*. Two approaches have been implemented. For minimizing the infrastructure requirements, Forby can rely on existing servers to maintain this information – either a CVS/svn server used by the group or an email account supporting IMAP access (e.g. Gmail). If no remote server is available, peers can be organized into a DHT (Pastry [15] in our current prototype), where one node is responsible for maintaining the active participants of a given group.

After obtaining the list of participants, the new node enters the dissemination tree and starts receiving, processing and propagating events, as any other node.

For small groups of users with no or minimal networking knowledge, private networks are a problem as they reduce the connectivity among computers. This is important as users' computers are increasingly in private networks, either because most organizations internally structure their network into private networks or because access is usually mediated by a router (or firewall), even at home.

Forby addresses this problem by organizing private and public nodes in separate trees. Public nodes create a public dissemination tree as stated earlier. Private nodes follow a different approach. The first node that connects to the system becomes the private network leader and tries to establish a connection with a public node. The other nodes that join that network connect to the leader, forming a private dissemination tree. When an event is disseminated, it is forwarded through the tree and when it reaches the leader, it is sent to the public peer, which will forward it in the public tree.

Any public node that receives an event, forwards to the other public nodes and to all private nodes connected to it, so that every message reaches all group nodes. Public nodes are used as a bridge between private networks, allowing private networks to receive notifications. Figure 4 shows the organization of public and private nodes.

All disseminated events are stored in event repositories for later use. Since Forby follows a best effort dissemination policy (which offers no guarantee that a message immediately reaches all nodes), this is necessary to provide fault-tolerance. Additionally, it is also used to provide support for disconnected nodes.

Event repositories can be local to each peer or they can be set in a remote server. Once again, as for the *rendez vous* point, a CVS/svn or IMAP server can be used, thus imposing no specific infrastructure requirement for running Forby. Besides simplifying the system deployment, this approach allows each system/group of users to define the configuration that better suits their environment/requirements.

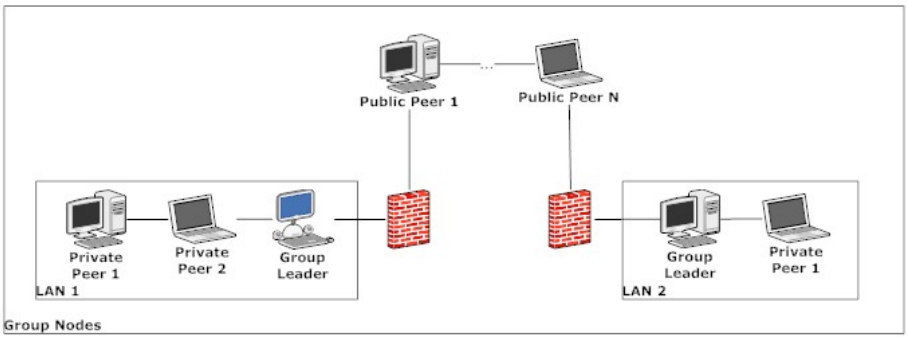

Fig. 4. Organization of private and public networks.

The use of a remote repository is also used to allow private networks to communicate with each other when no public node is available. In this case, the leader of each private sub-network periodically polls the server to check if new events are available and if they are, the leader forwards them to the other nodes in the network.

The dissemination system can be used to disseminate FSEvents or any other event that the application needs to propagate to other nodes. In any case, the disseminated event, *DisseminationEvent*, contains not only the event to be propagated but also all the information needed by the dissemination process (including the event sender and the sequence number). Each node keeps a summary of the received events (in a version vector) and periodically exchanges these summaries with neighbors (in the dissemination tree). A node can discover that it missed some event when exchanging these summaries or when it receives a new event (by discovering that some event is missing in the sequence of events from a node). In this case, the node can contact the remote repository — if one is being used — or disseminate an event request message, which will be replied by any node that has the requested event cached or locally stored.

For optimizing the event dissemination process, applications can define when an event makes a previous event obsolete – e.g. an event stating that a file has been modified usually makes a previous event with the same information from the same user obsolete. In this case, if the older event has not been propagated to all nodes, only the most recent event will be propagated.

Also, obsolete events can be deleted from the repositories and, when a node requests an event or a sequence of missing events, it needs only to receive the most up-to-date event that is not obsolete, optimizing the bandwidth and storage usage, and increasing the efficiency. Applications must define the rules used by the system to state that an event is obsolete. To simplify this process, the programmer can rely on base definitions or create a new one.

Finally, another feature offered by Forby's dissemination system is the possibility to send messages to a node using the reverse path followed by an event. This way, nodes can contact each other directly. As exemplified in the imple-

mented applications, this is very useful for allowing a node to contact the original node to obtain additional information, when necessary.

5 Application development

Forby can be easily used by any application that needs to provide groupware features relying on the information obtained by capturing and disseminating file system events. It allows programmers to focus on the application itself, rather then spending time dealing with the the details of efficient event capturing and dissemination. For using Forby, the application must set up the Monitoring and the Dissemination modules, by selecting the adequate parameters.

In the remaining of this section we detail three applications implemented using Forby. These applications have different goals – from providing awareness to a simple voting application, thus allowing to illustrate how Forby and its mechanisms can be used for supporting the development of different distributed groupware applications.

5.1 Awareness for version-control systems

In client-server version control systems (e.g. CVS, subversion), users checkout each file they are interested on and create a local copy of the file. Then, they can modify the files in isolation for how long they want. Finally, when the user decides that she has finished her changes, she submits the new version back to the server. This approach may lead to concurrent changes, as more than one user may change the same file concurrently without knowing about the actions of other users.

This is a typical scenario where it would be interesting to provide additional awareness to users. Several solutions have been proposed to address this problem (e.g.[14, 8, 16, 3, 9]). However, the proposed solutions either require the user to use an additional application or are defined as a plug-in for some specific text editor, thus limiting its use.

Without changing the editor, a possible solution for providing additional awareness information is the following. Whenever a user accesses a file, by opening and reading it in any application, the user must be informed if a new version exists in the server or if some user has modified his local uncommitted copy of the file. Whenever a user is accessing a file and some other user concurrently modifies the file, the user is informed about the possible conflict. In any case, the user should have the possibility to request the new version either by obtaining it from the server or by requesting the other user to commit the new version.

We have implemented this application using Forby, relying on the dissemination of events for file change notification. Thus, whenever, a user modifies a file locally, our system automatically disseminates to all other users an event that represents that modification – including information on the version being modified and on the user. When receiving this event, each peer will store the

event for possible future notification. If the file has been accessed recently, the user is also informed of the possible conflict.

Whenever a user accesses a file, by opening and reading it in any application, our system checks the events stored locally and determines if some user has modified her local copy (without propagating it to the server). Whenever this happens, the user is notified – figure 5 shows the notification presented when a user opens a file that is concurrently being modified (in this case, the user is using the *gedit* application, but this would work with any application).

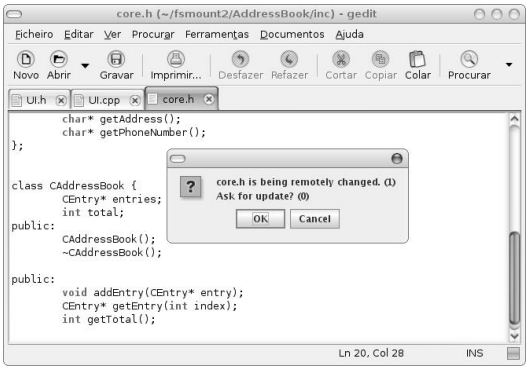

Fig. 5. User being alerted that the file she is opening has been modified by some other user.

When compared with the mentioned solutions that provide similar functionality, our approach has the advantage of working with any editor application, thus allowing users to continue using their preferred editors. Moreover, when compared with solutions that use additional applications, our solution allows the user to obtain the awareness information without having to change application – a simple pop-up is shown to users.

To implement this application in Forby, we have used the functionalities of the Monitoring and Dissemination modules.

The Monitoring module is used to monitor the file system activity and receive the needed events. In this case, we were interested in knowing, for a subset of the files in the file system, when a file had been accessed and when it had been modified. To this end, we have used the base filter that reports file changes and file accesses, with the change that file accesses are reported on the first read operation. On a file change, the application notifies other peers that a file has been locally modified. On a file access, the application uses this information to verify if the file has been remotely modified and to notify the user if necessary.

When deploying our solution, an adequate file system monitor should be used in each node, depending on the operating system.

The Dissemination module is mainly used to disseminate events about file modification by some user. To use it efficiently, the programmer must define

when an event make others obsolete. In this case, when two file write events are on the same file and user, the older event is obsolete. Thus, in the worst case, for each file, the dissemination system will only have to keep one event for each user, even if a large number of updates are executed by that user.

When deploying our solution, it is necessary to configure the adequate repository for *rendez vous* and event storage. By default, *rendez vous* relies on the CVS server to store the needed information. For event storage, local storage is the default option but an external repository can be used if the network configuration requires that.

For implementing the requests for commit, our solution sends a message to the peer that has the new version using the reverse path for the update message.

We had previously implemented a similar solution without relying on Forby [13] and using the version control system to propagate events among users. When compared with that solution, our solution based on Forby is much simpler, as Forby handle the details related with file monitoring and event dissemination. Additionally, the system can be used without modification in any platform that runs Forby (Windows and Linux, at the current moment).

5.2 Automatic directory replication

In this section we describe an application that automatically maintains several replicas of a directory hierarchy up-to-date. Our intended use was to allow a family to cooperatively maintain the set of family photos, by allowing each family member to immediately access photos added by the other members[1]. However, our approach could also be used to maintain several replicas of a workspace up-to-date.

As in the previous example, we have implemented this application using Forby, relying on the dissemination of events for notifying workspace changes. Thus, whenever, a file is created, removed or updated, our system automatically disseminates this information to all other peers.

When receiving a file creation or file update event, the peer automatically downloads the file from the source peer[2]. If the peer has a public network address, a direct connection is used. Otherwise, the application relies on the dissemination system to communicate with the source peer. When receiving a file deletion event, the peer removes the file locally.

All the replication process is automatic, without need for the application to explicitly notify the users. For the targeted environment, we do not expect concurrent updates to occur, thus a simple conflict resolution mechanism is used to guarantee that all replicas eventually converge to the same state. When receiving a file update (or file delete), the correspondent action is executed only if no more

[1] When compared with web-based services that provide similar functionality, our approach provides additional privacy as photos are not stored at some third-party server.

[2] For simplicity, we omit details related with configuration for delayed download or partial workspace replication.

recent event on the same file has been received (with the recent relation defined using the information used by the dissemination system). This guarantees that all peers will download the version that corresponds to the most recent event.

To implement this workspace replication mechanism using Forby, our application uses the Monitoring module to collect the information about file creation, deletion and modification. To this end, one of the basic event filter sub-modules has been used.

Regarding the use of the Dissemination module, it is necessary to define the obsoleteness rule. In this case, for some given file, a file update makes obsolete a file create and a file delete makes obsolete a file create and a file update from the same user.

Again, as in the previous example, when deploying the application it is necessary to decide which file system monitor, *rendez vous* and event storage mechanism will better fit the users environment.

5.3 Voting application

The third application we present is a very simple voting system. In this application, a user in the group may start a new voting process. For each voting, a ballot document with some pre-defined text format is created in each peer involved in the voting process. The user can cast his vote by modifying the ballot document.

When implementing this application using Forby, the following approach is used. For starting a new voting process, a user must create a voting request file with a pre-defined extension. The file, with some specific pre-defined format, includes the question to be posed, voting possibilities and a flag stating that the voting process should be active. Currently, the voting request file is usually created by copy and paste from given templates, but this functionality could be implemented in existing or in a new applications.

When the application detects that a voting request file has been set to active (by changing the flag in the voting request file), it uses the dissemination system to propagate to other peers the voting request. Upon reception of a voting request, a ballot document is created and a notification is presented to the user. The user could later cast his vote by modifying the ballot document file stored in the file system. Upon completion of his vote – marked using a completed flag, the application propagates the ballot to the original site, where the final decision is computed when some minimum number of votes were received.

To implement this application using Forby, our application uses the Monitoring module to know when a voting request or ballot has been modified. By verifying the active (resp. completed) flag in the voting request (resp. ballot) file, the application knows that it is time to propagate a voting request event (resp. ballot event). The event filter sub-module used by this application only propagates to the application the notification that a file has been modified.

Regarding the use of the Dissemination module, the same obsoleteness rules of the replication application are used. Again, as in the previous examples, when

	Execution time (ms)		
File size	No monitoring	JNotify	Minifilter (IFS)
500 KB	17	18	17
1 MB	38	38	38
10 MB	485	485	491
20 MB	966	990	977

Table 1. Average execution time for write operations.

	Execution time (ms)		
File size	No monitoring	JNotify	Minifilter (IFS)
500 KB	1	1	1
1 MB	3	3	3
10 MB	39	39	40
20 MB	80	81	84

Table 2. Average execution time for read operations.

deploying the application it is necessary to decide which file system monitor, *rendez vous* and event storage mechanism will better fit the users environment.

6 Performance Evaluation

Forby's event capturing relies on monitoring mechanisms that could possibly introduce some file system operation overhead. To analyze the relevance of that potential overhead, a simple experiment consisting on a set of file access operations was run. The experiment consisted on measuring the overhead, averaged over 500 repetitions, for read and write operations over different files and file sizes. Tables 1 and 2 show the resulting average execution time for the different monitoring tools.

The values on tables 1 and 2 show for both read and write operations a very low overhead. This low overhead is barely measurable and is probably more influenced by the increase in the overall system load than the potential delay, related to the registering of notifications, induced on the actual kernel operations. Moreover, the expected deployment environment consists on interactive user usage, in which this overhead is not observable in practice.

The low overhead is a noticeable benefit of kernel support for file system event notification, since, in contrast, file system call interception mechanisms induce observable delays [13].

7 Final remarks

In this paper we present Forby, a system that leverages modern lightweight mechanisms for file system activity monitoring with distributed event dissemi-

nation to provide groupware features to distributed users. As such, the system is composed of two main modules. The monitoring module allows applications to obtain the file system events they are interested on using a high level, operating system independent, interface. The dissemination module allows to disseminate events among groups of peers, handling all details related with reliability (including persistence and disconnection support). The dissemination module was designed to impose minimal infrastructure requirements by allowing to use existing servers (CVS/svn servers or email servers – e.g. Gmail). This simplifies the use of the system by any user.

The examples presented in the previous section show three different uses of Forby. The first solution provides awareness information for users editing data managed by a version control system. The second solution provides automatic shared data replication. The third solution implements a voting system. These examples illustrate that Forby can be used to develop different groupware features to distributed users. One important feature of these solutions, and of solutions that rely on file system mechanisms, is that it allows providing groupware features to any application, allowing users to continue using their preferred applications.

However, it is important to understand that these mechanisms can only act when user applications access the file system. This makes these solutions better suited for providing groupware features in non-realtime (or asynchronous) settings. This limitation can be alleviated in some applications, by setting the auto-save period to a short time, thus allowing the system to have knowledge about user actions very quickly.

As mentioned earlier, we believe that combining the use of file system monitoring and distributed event dissemination has the potential to allow providing groupware features to users using unmodified applications. We see this approach not as a replacement of other alternatives, but rather as a complement.

References

1. Daniel Avrahami and Scott E. Hudson. Qna: augmenting an instant messaging client to balance user responsiveness and performance. In *CSCW '04: Proc. of the 2004 ACM conference on Computer supported cooperative work*, pages 515–518, New York, NY, USA, 2004. ACM.
2. James Begole, Mary Beth Rosson, and Clifford A. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Trans. Comput.-Hum. Interact.*, 6(2):95–132, 1999.
3. Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proc. of the European Conference on Computer-Supported Cooperative Work - ECSCW'07*, pages 159–178, 2007.
4. Paul Dourish, W. Keith Edwards, Anthony LaMarca, John Lamping, Karin Petersen, Michael Salisbury, Douglas B. Terry, and James Thornton. Extending document management systems with user-specific active properties. *ACM Trans. Inf. Syst.*, 18(2):140–170, 2000.

5. G. Fitzpatrick, P. Marshall, and A. Phillips. CVS integration with notification and chat: lightweight software team collaboration. *Proc. of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 49–58, 2006.
6. Tom Gross and Wolfgang Prinz. Awareness in context: a light-weight approach. In *ECSCW'03: Proc. of the eighth conference on European Conference on Computer Supported Cooperative Work*, pages 295–314, Norwell, MA, USA, 2003. Kluwer Academic Publishers.
7. Susanne Hupfer, Li-Te Cheng, Steven Ross, and John Patterson. Introducing collaboration into an application development environment. In *CSCW '04: Proc. of the 2004 ACM conference on Computer supported cooperative work*, pages 21–24, 2004.
8. Claudia-Lavinia Ignat and Gérald Oster. Awareness of Concurrent Changes in Distributed Software Development. In *Proc. of the International Conference on Cooperative Information Systems (CoopIS'08)*, pages 456–464, Monterrey, Mexico, November 2008.
9. Claudia-Lavinia Ignat, Stavroula Papadopoulou, Gérald Oster, and Moira C. Norrie. Providing Awareness in Multi-synchronous Collaboration Without Compromising Privacy. In *Proc. of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2008*, pages 659–668, San Diego, California, USA, November 2008.
10. JNotify. Jnotify linux api, 2007. http://jnotify.sourceforge.net/.
11. Du Li and Rui Li. Transparent sharing and interoperation of heterogeneous single-user applications. In *CSCW '02: Proc. of the 2002 ACM conference on Computer supported cooperative work*, pages 246–255, New York, NY, USA, 2002.
12. Du Li and Jiajun Lu. A lightweight approach to transparent sharing of familiar single-user editors. In *CSCW '06: Proc. of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 139–148, New York, NY, USA, 2006.
13. D. Machado, N. Preguiça, C. Baquero, and J. Legatheaux Martins. Vc2 - providing awareness in off-the-shelf versioncontrol systems. In *IWCES9: Proc. of the Nineth International Workshop on Collaborative Editing Systems*. IEEE Computer Society, 11 2007.
14. 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.
15. Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.
16. Anita Sarma, Zahra Noroozi, and Andre van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proc. of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.
17. C.R.B. De Souza, S.D. Basaveswara, and D.F. Redmiles. Supporting global software development with event notification servers. In *Proc. of the ICSE 2002 International Workshop on Global Software Development*, 2002.
18. Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, 2006.
19. E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.