ABSTRACT
For many years, people have wanted to share single-user applications. The vision has been to replicate instances of a single-user application across the network and transmit input events from one instance to the others. While there have been attempts to build such a system, they have had limited success due to an implicit assumption of application determinism. This assumption is untrue in the face of environmental differences among computers running the applications. After describing one approach to solving this problem, we discuss an alternate approach we are implementing in our Zipper system which addresses several shortcomings in the prior work. We conclude by briefly discussing several of the ongoing research issues with our prototype.

ACM Classification: H.4.3 [Information Systems Applications]: Communications Applications—computer conferencing, teleconferencing, and videoconferencing. H.5.3 [Information Interfaces and Presentation]: Group and Organization Interfaces—computer-supported cooperative work, synchronous interaction.

General terms: Design, Theory.

Keywords: Replicated application sharing, determinism, externalities, aspect-oriented programming.

INTRODUCTION
Today, the choices for synchronous sharing of applications are limited. A few applications are developed with integrated collaborative features, but often these applications lack other features found in their single-user counterparts. And even though there are some counter examples, the vast majority of end-user applications are written as single-user applications. Collaborative application features are difficult to write and are not high on the list of development priorities. Users, who may be motivated to add collaborative features to an application, often lack the resources (e.g., the source code) to accomplish this task. To share applications, users have had to resort to “screen-scripting” techniques using tools such as IBM Lotus Sametime [19]. These tools track changes to a computer’s screen buffer and transmit the changes as bitmaps to the other collaborators. Although able to share many applications, this approach is both CPU- and network-intensive, limiting its utility.

Recognizing the value of sharing single-user applications in a more flexible manner, we revisit the technique of replicated application sharing. In replicated application sharing, separate copies of a single-user application are run on each collaborator’s computer. Events from one copy (e.g., keystrokes) are broadcast to the other copies where they are processed as if they had been generated locally, allowing the distributed applications to stay synchronized. Such collaborative use of replicated, single-user applications has long been a dream of CSCW practitioners. If such a system were available, then the myriad of single-user applications could be repurposed as collaborative tools. Not only would people be able to collaborate, they would be able to collaborate with the applications that are most familiar.

In this paper, we first identify the key issues we are addressing. A consideration of alternative architectures and the issues of state consistency and performance motivate our adoption of a hybrid architecture. Next, we discuss our approach and how it differs from prior work, particularly with regard to its support for both strict and relaxed WYSIWIS (What-You-See-Is-What-I-See) and how it handles externalities which cause consistency problems. After that, we discuss our prototype which makes extensive use of aspect-oriented programming to add collaborative capabilities to an existing user interface framework. We then discuss our findings, focusing on the suitability of aspect-oriented programming for sharing. We conclude with observations and areas of ongoing work with our Zipper prototype.

PROBLEM STATEMENT
The fundamental question we are trying to answer is what a user interface framework would need to be like to support replicated application sharing. Let’s start by considering alternate architectures for application sharing. Let’s start by considering alternate architectures for application sharing.
Patterson [18] has described several architectures for synchronous groupware applications. All of the architectures start with the simplified event processing common to most applications shown in Figure 1. A user input event, $e_x$, is delivered to the application. Some processing occurs which modifies the underlying model. The view reflects the changes to the model and updates the display producing some output, $o_x$. Often, the application will need to access some external resources such as the file system or a database. These resources, which are not a part of the application, are located in the environment.

The various architectures for sharing differ on two key issues: 1) how they provide for application consistency and synchronization, and 2) how flexible they are with respect to allowing collaborators to have differing experiences with the software. Dewan and Choudhary have referred to this flexibility as “flexible coupling” [7]. This section will present several of the options and discuss their relative merits with respect to consistency and flexibility. In particular, we focus on architectures for supporting collaboration transparent sharing—sharing an application that was not written to be collaborative.

Centralized Architectures

Centralized approaches to application sharing have the feature that there is a single copy of a portion of the system which is shared among collaborators. Note that “centralized” in this case does not imply a networking architecture (i.e., client-server versus peer-to-peer). Rather, it means that state consistency in these systems is ensured by having all collaborators access and use the same state.

Screen sharing systems such as IBM Lotus Sametime [19], Microsoft NetMeeting [17], and VNC [22] work by replicating the display and centralizing the view, model, and environmental resources. This is shown in Figure 2. Maintaining display synchronization in this case is straightforward as there is a single copy of the application that all users manipulate. In fact, the products in this space demonstrate their utility for providing collaboration transparent sharing of any number of end-user applications. However, there are several drawbacks to this approach. First, it enforces strict WYSIWIS sharing among the collaborators. There is no way, for example, for collaborators to scroll a window separately, though research has shown the need for more relaxed forms of WYSIWIS under certain circumstances [20]. Second, it is inefficient since large bitmaps are sent across the network continually. The latencies between one user performing an action and the other users seeing the results of that action can be quite large.

Replicated Architectures

Replicated approaches to application sharing have the feature that there are multiple copies of the entire application distributed within the network. Instead of relying on shared state to maintain consistency, replicated application sharing uses a state synchronization mechanism, such as turn-taking or operational transformation [21], to ensure that the application state is consistent across all replicas.

On the face of it, the idea behind replicated application sharing is simple. If all of the collaborators have a copy of the same single-user application, then one user can “drive” all the application replicas (this user is referred to as the “moderator” and is said to have the “floor” when interacting with the application). Underlying this idea is the notion that, if the same sequence of events (e.g., user input) is sent to replicated instances of the application, then the application state will be manipulated and modified in the same manner in each of the application copies and each collaborator will see the same result. This can be seen graphically in Figure 3. This approach is much more network-efficient than centralized systems, since the bandwidth of the input events is small compared to the application output that gets displayed to the user.

When the moderator interacts with the application, an input event is intercepted and forwarded to the other collaborators in addition to being executed locally. This is shown as the dotted line between the applications in Figure 3. Ap-
Application logic will cause some sequence of operations as a result. The processing will wind through the application and eventually produce some output. In parallel, the event is interpreted by the remote collaborators. Assuming that the applications were in the same initial state, the same processing flow occurs and the same output is produced.

To prevent cycles, it is critical to differentiate between user-generated events and injected events; events injected into the application should not be intercepted and forwarded to the other collaborators.

Relaxed WYSIWIS

Because replicated application sharing is dealing with higher-level application events, and not simply bitmaps and mouse movements as in screen sharing, the possibility exists for more selective event sharing or relaxed WYSIWIS. Systems like CoWord [23] share only changes to the application model and rely on operational transformation to maintain consistency. Users can scroll separately while the system ensures that edits made by one user are consistently merged with those of the other users. In the extreme, the collaborators can see totally different versions of the same underlying model. Li and Li [15], for example, have demonstrated different editors, GVim and Microsoft Word, editing the same underlying document. One drawback of this approach is that, while it allows for relaxed WYSIWIS, it is not possible to provide strict WYSIWIS. Operational transformation focuses on the model; since view information is not in the model, it is not possible to share view-related information (e.g., scrolling or selecting). This could be a problem in certain domains such as remote debugging or distributed learning where it is important to have collaborators see the same interface.

Other replicated application sharing systems have taken a more liberal view on what can be shared [10], and allow both the model as well as portions of the view to be shared. These systems typically rely on some sort of floor control mechanism, either explicit or implicit, to limit the opportunities for conflicting user input.

Externalities

MMConf [6], Dialogo [13] [14], and the first Rapport system [1] were shared windowing systems which captured windowing system input events and transmitted those events to application replicas. However, all of these systems ran into synchronization problems and the replicated applications would display different output. The problem with the replicated application sharing vision is that it assumes state changes within the application are deterministic. In replicated application sharing, that assumption is usually incorrect.

This problem is illustrated in Figure 4. In this example, the application logic consults something in the environment outside of the application (shown as the lines to and from the “environment” box). This could be accessing a file on the local file system or making a system call to retrieve the time. Since the environments on the two computers are not identical, the environmental access returns a different result. Subsequent processing follows a different path through the two applications. As a result, the collaborators see different outputs.

Crowley, et al. talk about four impediments to maintaining state: “differences in initial application state, misordered input events, nondeterministic applications, and latecomers” [6]. Lauwers, et al. make a point of noting that “the synchronization problem is tractable when the shared applications are deterministic” [14]. They claim further that an application is deterministic if, starting from the same initial state, the application will generate the same sequence of outputs given the same sequence of inputs. In addition, the application output cannot depend upon the timing between input events. Ahuja, et al., describe the problem with applications that utilize local state and say that “the maintenance of this environmental consistency is not generally possible” [1].

Many seemingly innocuous items in the environment can cause problems. Typical of such items are preference files (e.g., retrieving a user’s preferred font or foreground color)
and operating system calls to retrieve the current system
time, IP address, or next random number. Begole, et al.,
call these environmental problems externalities. More spe-
cifically, they define an externality as an input (other than
from the user) or an output (other than to the display) that
is external to the application itself [2]. Note that central-
ized approaches to application sharing do not have this
problem since there is only one copy of the application and
it only has access to one environment (see Figure 2).

Hybrid Architectures
The key to fixing the consistency problem with replicated
architectures is what we call environmental replication—
replicating environmental state across the distributed applica-
tions. Strict environmental replication is not possible;
every computer is unique. Rather, it is important to try to
minimize the likelihood of externalities and, when possible,
to detect and correct the ones that cannot be prevented.
One approach to minimizing externalities is to ensure that
there is a common operating environment for all of the
collaborators. Java, for example, simplifies the process of
replicating applications since it defines a virtual machine
which provides abstractions for many elements in the oper-
ating environment. However, with different versions of the
Java virtual machine and many core Java methods imple-
mented using native operating systems calls, Java by itself
does not solve the problem. A way to share the results of
externalities is also needed. In essence, this is Patterson’s
“hybrid architecture” [18], but he was not addressing the
problem of externalities directly. The key to the hybrid
approach is to replicate deterministic operations and to
centralize operations that cause externalities.

![Figure 5. Flexible JAMM’s Proxied Externalities.](image)

Flexible JAMM is perhaps the best example of this ap-
proach [2][3]. Flexible JAMM improved upon earlier win-
don-sharing systems by exploiting properties of the Java
language to dynamically replace single-user components
with specially-written multi-user counterparts. This proved
more successful than window sharing systems in that it was
able to handle more externalities. In particular, they noted
that any Java object that is implemented using native calls
to the operating systems is very likely to introduce an ex-
ternality. Flexible JAMM replaced calls to these objects
with calls to a proxy and wrapped the actual object with a
small server. The proxy object would use Java Remote
Method Invocation (RMI) to access the server and retrieve
the necessary data. In this way, all collaborators would see
the same result to the externality. This is shown in Figure
5. Since all collaborators accessed the same environmental
state via the server, all would see the same result and the
applications would remain synchronized.

There are limitations to the Flexible JAMM approach.
First, the proxied externality adds a server to an otherwise
distributed, replicated architecture (Begole, et al., called
this a “semi-replicated” architecture [2]). This both de-
creased fault tolerance and increased latencies. Second,
care must be taken with environmental calls that can pro-
duce different results each time they are invoked (e.g.,
calls to the system time or to a random number generator).
In these cases, it is important to share the result of a single
call and not allow multiple accesses to the environment.
In other cases, it may be neither necessary nor desirable to
effectively duplicate externalities across application replicas
[14][2]. Indeed, the operating environment for each replica
should be the same. Third, there are additional
security concerns with having multiple servers running
which provide access to pieces of the operating environ-
ment. Finally, the replacement of Java objects at run time
itself has several limitations (e.g., sub-classes of replace-
able classes cannot, in general, be replaced).

THE ZIPPER APPROACH
We wanted a system with the consistency of a centralized
approach and the performance and flexible sharing of a
replicated approach. Since we view performance and flexi-
bility as critical to the collaborative experience, our re-
search has focused on fixing the externality problem with
replicated application sharing via a hybrid approach. Zip-
per differs in two significant ways from prior work on rep-
licated application sharing.

First, we have extended the notion of relaxed WYSIWIS to
flexible WYSIWIS. Flexible WYSIWIS recognizes that
there are times when literal WYSIWIS is critical, e.g., for
online training or debugging. However, there are also
times when the only critical state to share is the application
model and users should be able to manipulate their applica-
tion views independently. To support this, Zipper has de-
veloped the notion of zipping and unzipping. When a
shared application is totally zipped, collaborators have
strict WYSIWIS. When the application is totally unzipped,
only the important state in the application model is shared.

Second, we introduce the notion of externality forwarding.
With externality forwarding, in addition to sending state-
changing events to remote collaborators, we also detect and
send the results of application externalities. In this manner,
we have the benefit of centralized access to environmental

---

state without the overhead of a round-trip communication to a server.

**Flexible WYSIWIS: Zipping and Unzipping**

Graphical applications are typically implemented with a model-view-controller paradigm. User input is received by the controller, which causes the model to be modified or updated in some manner. The view then reflects the changes to the underlying model on the user’s display.

In reality, certain view-controller interactions never touch the model. For example, when a user manipulates a scroll bar, the contents of a viewport get updated to reflect the new scroll position. In most cases, though, the model is not involved with this interaction. In essence, there is state that we would like to share (e.g., scroll position) that is not reflected in the application model. This is demonstrated in Figure 6 which shows the input processing and the output rendering for a simple text editing application. When the user types, the keystrokes are received by the controller and the underlying model, the text of the document, is updated. The updated model is then rendered for display to the user. If the user scrolls the text editor window, the scroll events are again received by the controller. This time, however, the model is not affected. Instead, the coordinates of the viewport are updated and the display is redrawn. Similarly, if the user selects some text by sweeping it out with the mouse, the effect is to set a text range which is handled differently during the display rendering (e.g., the selected text is highlighted by drawing it with foreground and background colors inverted).

The key to flexible WYSIWIS is the realization that these events can be treated differentially. The editing events affect the model and need to be forwarded to all collaborators. If the scrolling events are forwarded to the other collaborators, users will experience synchronized scrolling. If the selection events are forwarded to all collaborators, then all users will experience synchronized selection. The scrolling and selection events only affect the view, however; if we do not forward them, then users experience relaxed WYSIWIS where only the model is being shared. If we forward one or the other of them, then we have achieved a level of flexibility not possible with prior application sharing systems.

Assuming that we start a collaborative session with the application fully zipped, unzipping it is straightforward—we just stop forwarding the unzipped events to the other collaborators. Rezipping the application is harder. In this case, the system must first synchronize the collaborators’ divergent states. For example, if we want to again share scrolling events, everyone must be brought to a consistent scroll position before new scroll events can be shared. For certain types of events, such as scrolling, this is relatively straightforward—choose one collaborator, grab his current scroll position, and send that position to all collaborators. In the general case, however, this is similar to bringing a latecomer to the collaboration up to speed (except that the application models will already be synchronized).

**Externality Forwarding**

Our concept for handling externalities is similar to that of Begole, et al., but our approach is different. First, rather than rely on a client-server approach to distributing externalities, our plan is to forward the results of externalities to the remote collaborators, eliminating the network round trip. Second, rather than write a custom class loader and multi-user replacements for standard classes, we plan to use aspect-oriented programming to “hook” single-user applications in a transparent way.
package the results of the call and send them to the other collaborators. When the remote collaborator’s application gets to the point of the externality, it must check to see if it has already received the result from the moderator. If the result has not been received, the application must wait. Once the result has been received, the application is able to proceed and produce the correct output. This is illustrated in Figure 7.

There are challenges with this approach. For example, what if the result of an externality is a pointer, such as a reference to a file? The reference would have no meaning in the context of the other collaborators. Instead, we need to dig deeper to get the “result.” In the case of the file pointer, for example, we are not really interested in the bytes which get read from the file. These represent a “result” which we can forward to the other collaborators. The research challenge is to choose appropriate externalities to forward.

Aspect-Oriented Programming

One of the recent advances in software engineering has been the development of aspect-oriented programming (AOP) [12]. “Aspects” are special objects that define rules for actions occurring before, after, and within code. While object-oriented programming is a methodology for software modularization, separating specific pieces of application functionality into objects, AOP extends this separation further by effectively modularizing calls that are repeated across disparate objects.

The canonical example for AOP is application logging. Objects in an application make calls to a logging object throughout their methods. This results in logging code being repeated and mixed with core functionality. The AOP approach would replace all of the logging calls with a logging aspect that defines rules to ensure logging occurs at the appropriate time and in the appropriate objects. The objects themselves have no knowledge they are being logged, and all logging functionality is centralized in a single aspect.

We use AOP for “hooking” the state-changing events of single-user applications in order to make them multi-user [4][5]. Perhaps the biggest appeal of AOP for collaborative applications is that the applications being made collaborative do not need to be modified or even recompiled. Other than knowing the places to hook them (called join points in AOP terminology), we need very little information about the applications to make them collaborative. AOP has proven more flexible than the dynamic class replacement used in Flexible JAMM. First, an AOP approach is not limited to sharing only events from “well-behaved” toolkits like Swing. Since any method can be hooked, we can capture events from any library, even those without a clean model-view separation. Second, a subclass of a shared object will inherit the sharing of the superclass provided by AOP. This was a problem with Flexible JAMM [3].

IMPLEMENTATION

For the past year, we have been implementing a prototype Zipper system for collaboration-transparent, replicated application sharing.

Environment

For the prototype, we have chosen to limit our applications to those that are implemented on top of the Eclipse [8] framework. We chose to constrain our prototype efforts to Eclipse for several reasons. First, Eclipse provides a Java operating environment. As mentioned earlier, the Java virtual machine abstracts many operating system dependencies which could cause externalities. Second, Eclipse has well-defined graphics and widgets libraries (e.g., JFace and GEF) which, like Java Swing, provide a nice model-view separation. Third, Eclipse is open source so we have access to the source code for help in determining where the framework should be “hooked” to capture events. Fourth, Eclipse is cross platform, so we have the opportunity to try out Zipper application sharing across various hardware platforms. Finally, various efforts within the Eclipse community are aimed at making it a better platform for development of rich client applications; we anticipate that an increasing number of end-user applications will be deployed on top of Eclipse in the future.

Zipper is implemented as an Eclipse feature, a set of related Eclipse plugins which can be added to an Eclipse installation. This aids with deployment. In fact, since no underlying Eclipse code was changed, users need only install the Zipper feature into their Eclipse environment in order to start collaborating. Initially, we targeted text editing applications in Eclipse, which included the Java editor. Our thinking was that developers could use Zipper to collaborate on their code development. More recently, we have begun to look at different classes of applications. For example, we have looked at applications built using the Eclipse Graphical Editing Framework (GEF) including a task editor and a digital logic editor. By experimenting with a variety of different applications based upon different libraries, we hope to create abstract event and externality handlers that cut across a wide variety of applications.

As mentioned earlier, our focus on replicated application sharing does not imply any particular network architecture. In fact, we have developed a high-level, protocol-independent communications interface. We have two separate implementations of this interface, selectable via a command-line parameter. One uses a home-grown client-server system for real-time notifications; the other uses reliable IP multicasting provided by JGroups [11] for true peer-to-peer operation.

We have also been using Lippert’s AspectJ-Enabled Eclipse Runtime (AJEER) [16] to weave aspects into the Eclipse framework. AJEER is an Eclipse plugin that allows aspects to be contributed to other Eclipse plugins. AJEER performs load-time weaving of aspects, so there is no need to recompile the plugins to which the aspects are being added.
Operation
Figure 8 shows the Zipper prototype. Two separate instances of Eclipse are shown, representing two different users. Each user has a different Eclipse perspective which is showing different combinations of Eclipse views. The file “Main.java” is being shared in the Java editor. To start sharing, a user simply brings up the context menu associated with an editor and selects the option for sharing. The user can choose one or more collaborators with whom to share the editor. The users who have been invited get a pop-up dialog asking if they’d like to join the collaborative session. If they choose yes, a new editor is created for them with the contents of the file being shared. Also, the originator gets a notification when they have joined. The background of the editor changes from white to a light green color indicating that the editor is being shared. (This change in background color is introduced without changing the base application through the use of an aspect.) In the figure, the collaborators have synchronized both their scrolling and their text selection.

Figure 9 shows a close up of the Zipper control panel view from Figure 8. A separate tab for controlling the level of sharing is created in the panel for each collaborative session. The name of the editor being shared is displayed on the tab; selecting different tabs will activate the corresponding editor. Each tab has two controls, one for inviting new participants and one for leaving the session. In addition, there are checkboxes that allow the collaborators to zip and unzip the application. As in the example in Figure 6, sharing a text editor shares three different types of events: editing (which includes typing, cutting, and pasting), selecting, and scrolling. There is no option for turning off the sharing of editing events since these events affect the underlying application model. Scrolling and selection, however, can be turned on and off independently. There is no explicit floor control in the prototype—any user can edit or turn scrolling and selection on or off. When a user turns scrolling or selecting on, his scroll position or text selection is sent to resynchronize all of the users.

KEY FINDINGS
In this section, we discuss several observations from our prototyping efforts. Some of these findings are issues with any replicated application sharing system. The others are
more particular to our Zipper system and its flexible WYSIWIS capability and aspect-oriented approach to adding collaboration to existing applications.

**Orthogonal Layers**

One surprising result, although it seems obvious in retrospect, is that application layers to share need not be in a strict hierarchy. Our thinking had been that sharing any particular event layer would necessarily imply sharing all of the layers below it. (In fact, the project name and zipper analogy come from this thought.) While that may be true in some cases (e.g., sharing a text selection implies sharing the underlying text), it is also the case that layers can be orthogonal. In our text editor, for example, scrolling can be shared independently of text selection and vice versa.

**Generality**

Creating the shared text editor demonstration was easier than we expected. In fact, although we concentrated on the Java editor, it turned out that we could easily hook events on a more general superclass which enabled us to share any text-based editor in Eclipse. However, our solution only works for JFace text objects. Although the concepts for other JFace objects are the same, we have been unable to reuse the aspects to the extent we had hoped. We would need additional hooks for sharing other JFace objects such as the tree and table viewers. Indeed, reusability of aspects is itself a topic of research [9].

GEF turns out to be an even harder problem. GEF is a fairly high-level framework. As such, it defines a few abstract classes and requires the developer to fill in the details. The base GEF objects have almost no functionality and so it is not possible to add aspects to those objects in any general manner. With text editors, we were at least able to tackle an entire class of problems; it appears at this time that it is not possible to make a similar claim for GEF. GEF-based editors would need to be handled on a case-by-case basis. So, while Zipper has sought generality, it has been difficult to achieve.

**Use of Aspect-Oriented Programming**

Aspects in this project have proven themselves most valuable in what we call “code archaeology”—getting a better understanding of the substantial Eclipse code base and determining which input events are of interest. The fact that we could add aspects to core Eclipse classes without recompiling made it even more attractive to use aspects in this manner.

Initially, we envisioned that we would use aspects for hooking the actual user input events to forward to the remote collaborators. As we learned more, though, we discovered that the Eclipse objects in which we were interested already had fairly complete APIs. Often, the trick was getting a handle to the appropriate object. For example, context menus in Eclipse are added to IEditorPart objects. So, when a user invokes the “share” menu, we know which IEditorPart was selected. However, an IEditorPart is really just a container; what we need is the ITextViewer object contained within it. The ITextViewer has interfaces for text modification, scrolling, and selection. Unfortunately, there is no method for determining which viewer is contained within a given editor. We ended up using aspects to add the missing API. When the user selects the “share” menu, we simply look up the corresponding ITextViewer and add our Zipper objects as listeners for changes to the text, selection, and scroll position.

We use aspects in several other unexpected ways. First, we use aspects to resolve issues with the way plugins get started within Eclipse. We cannot assume that our Zipper plugin will be started before editors that might be shared. So, we keep track of editor-viewer pairs as they are created with one aspect. We use another aspect which watches for the initialization of our Zipper plugin. When the Zipper plugin has been instantiated, we pass it the list of editor-viewer pairs which have already been created. Second, for our GEF experiments, we determined that GEF Command objects should be sent among collaborators. The Command class does not implement the Java Serializable interface, so we could not send them across the network. We use an aspect to add the Serializable interface to Command to get around this problem.

**Other Issues**

**Other Sources of Non-Determinism**

In addition to externalities, there are other sources of non-determinism in replicated applications. As others have reported, it is difficult, if not impossible, to maintain determinism if the timing of input events is critical. In addition, multithreaded applications can cause problems by allowing events to be generated in different sequences.

**Latecomers**

The issue of latecomers is a serious problem with replicated application sharing. In any realistic scenario, however, even starting a collaborative session is a latecomer problem. It is our goal to allow flexible collaboration so that a user need not think *a priori* about collaboration, but rather can collaborate when the need arises. As a result,
even the first collaborator to be invited is an example of a latecomer. We are investigating the use of state exchange via process migration to handle latecomers.

Providing Collaborative Features

We must also consider how to add collaborative features such as telepointers to the otherwise single-user applications. Aspects seem particularly well-suited to this task, as we have already demonstrated [5]. We must also have a mechanism for finding potential collaborators and starting sessions. We envision a separate application, perhaps tied in to a “buddy list” client, for finding potential collaborators and starting conferences with them.

DISCUSSION

As stated at the outset of this paper, our goal has been to investigate what elements would need to be in a user interface framework for supporting replicated application sharing. In particular, we approached this problem with the use of aspect-oriented programming. How successful have we been? What is the role of aspects in replicated application sharing?

On the one hand, there are those who would advocate that sharing is a canonical example of a “cross cutting” concern, which makes it perfect for aspect-oriented programming. But, as our experience has shown, the answer is not that clear. There are several issues with using aspect-oriented programming in this manner.

The biggest problem with the use of aspects has been finding the correct join points. We had access to the Eclipse source code; without detailed knowledge about an application’s internal structure, it would be difficult to find the appropriate join points. Decompilers and tracing programs can be useful when source code is not available. However, without clearly defined APIs on which aspects can be placed, join points feel very fragile. It is easy to put an aspect on an internal object, but if that object’s interface changes, the aspect stops working. While useful for experimentation, this is not viable for deployment.

Performance of aspect-oriented programming is also a concern. While aspect implementations continue to improve, they still impart some overhead on the system. Load-time weaving of aspects places a burden on run-time application performance. There are also deployment concerns as additional libraries may need to be included with an application, increasing on-disk size and download times. Perhaps most importantly, aspects may “cast too wide a net” and be executed for parts of the system never meant to be shared.

Although aspects have been extremely useful for our research, ultimately the aspects we have used point to deficiencies in the underlying user interface frameworks. JFace and GEF were just not written to be shared. We believe that to enable sharing and deal with problems such as externalities, the user interface framework needs to be designed and developed with sharing in mind. Does that mean that aspect-oriented programming is not useful for replicated application sharing? No—in fact, aspect-oriented programming might be the perfect technology for adding the cross-cutting concern of collaboration to a user interface library. We do argue, though, that the library needs to be designed in such a way that the use of aspects is not an afterthought that is subject to breaking when the interfaces of internal objects change. One of the issues for the aspect-oriented software development community is to develop design patterns for the use of aspects much the way that they have been developed for the object-oriented development community.

Of course, once we conclude that we need a new sharable UI framework, we can simply build the necessary capabilities into that framework. In this approach the aspects serve as an important investigatory tool for identifying the requirements of the new UI framework, but they will not remain as a part of the solution. Our ongoing work is investigating what it would take to develop such a collaboration-aware user interface framework based upon the JFace library. Such a framework would allow countless applications to be developed in a collaboration-transparent manner.

While a collaboration-aware user interface framework is necessary, it may not be sufficient. In particular, the framework may need to be accompanied by a set of environmental access tools with accompanying guidelines regarding best practices. To ensure that access to a set of properties does not cause an externality problem, for example, the developer needs to be encouraged to use the framework-supplied methods for accessing them. There is no way to prevent the developer from accessing the properties file directly, however. It is in these cases where externalities will continue to be a problem.

CONCLUSION

In this paper, we have discussed the vision of replicated application sharing. After first describing the vision, we explained the problem of environmental differences, or externalities, among application replicas. Although we are not the first to propose environmental replication as a solution to the problem, we feel our approach, based upon forwarding of results, is potentially more efficient. We also described our experiences using aspect-oriented programming as an approach for hooking events in an application. AOP also addressed some limitations in earlier work that relied on dynamic class replacement.

While AOP has been extremely useful for investigating and adapting existing UI frameworks to support replicated application sharing, it seems unlikely that this approach is fully tenable on its own. We will need a new UI framework that either supports the introduction of aspects as a cross-cutting concern or directly incorporates support for replicated sharing. Admittedly, many research questions remain, but the Zipper prototype has demonstrated some preliminary success and is revealing the issues that a UI framework designed for replicated sharing must address.

ACKNOWLEDGMENTS

We would like to acknowledge our intern, Adam Marcus, who implemented substantial pieces of the Zipper proto-
type including the abstract communications layer. We would also like to thank Martin Lippert for making AJEER available and answering numerous questions about the use of aspects.

REFERENCES