Another Look at Replicated Application Sharing

Steven L. Rohall and John F. Patterson
IBM T.J. Watson Research Center
One Rogers Street
Cambridge, MA 02142
+1 617 693 1840 +1 617 693 4236
firstname_lastname@us.ibm.com

ABSTRACT

For many years, people have wanted to share single-user applications. The vision has been to replicate instances of a single-user application throughout the network and transmit input events from one instance to the others. While there have been attempts to build such a system, they have not succeeded 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.

Categories and Subject Descriptors

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.

1. INTRODUCTION

Today, the choices for synchronous sharing of applications are limited. A few applications, such as multi-user games, are developed with integrated collaborative features. The vast majority of end-user applications, however, are written as single-user applications. Collaborative application features are difficult to write and are not often 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-scraping" techniques using tools such as IBM Lotus Sametime [11] and Microsoft NetMeeting [10]. These tools track changes to a computer's screen buffer and transmit the changes as bitmaps to the other collaborators. This is both CPU- and network-intensive, limiting this technique's 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. In this note, we will describe the vision of replicated application sharing and the reason why that vision has not been achieved—an implicit assumption of determinism. We will then discuss an attempt at fixing the determinism problem followed by our own approach to the problem in our Zipper system using a technique we call "externality forwarding." We conclude with a discussion of some of the remaining research issues with our Zipper prototype.

2. THE VISION

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 to which they are accustomed.

On the face of it, this idea is simple. If all of the collaborators have a copy of the 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 1. This approach is much more network-efficient than screen scraping systems, since the bandwidth of the input events is small compared to the application output which gets displayed to the user.

Figure 1 shows two collaborators (although theoretically there could be any number of collaborators); the collaborator on the left is the moderator interacting with the application while the collaborator on the right is observing the moderator's actions.

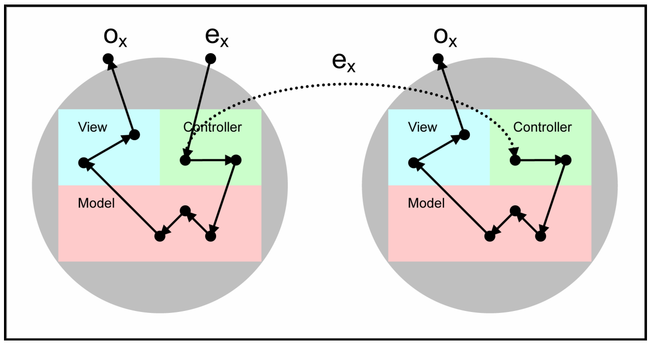

Figure 1. Replicated Application Sharing Vision

When the moderator interacts with the application, an input event, labeled e_x, is sent to the controller for interpretation. Application logic will cause some sequence of operations as a result. This processing flow is shown as the black lines. The processing will wind through the model and the view and eventually produce some output, o_x. In replicated application sharing, the initial event, e_x, is caught and transmitted to the other collaborators (shown as the dotted line in Figure 1). The event is then interpreted by the remote collaborators. Assuming that the applications were in the same initial state, the same processing flow occurs and the same output, o_x, is produced.

3. WHAT'S WRONG WITH THE VISION?

There have been numerous attempts to build replicated application sharing systems [6]. MMConf [5], Dialogo [8][9], and the first version of Rapport [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 where the replicated applications would be displaying different output. The problem with the vision is that it assumes state changes within the application are *deterministic*. In replicated application sharing, that assumption is usually incorrect.

Crowley, *et al.* talk about four impediments to maintaining state: "differences in initial application state, misordered input events, nondeterministic applications, and latecomers" [5]. Lauwers, *et al.* make a point of noting that "the synchronization problem is tractable when the shared applications are deterministic" [9]. 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].

This environmental problem is illustrated in Figure 2, which shows the same replicated application illustrated in Figure 1. In this example, however, 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, while one collaborator sees output o_x, the other collaborator sees output o_y. Begole, *et al.*, call these environmental problems *externalities*. More specifically, they define an externality as an input (other than the user) or an output (other than the display) that is external to the application itself [2].

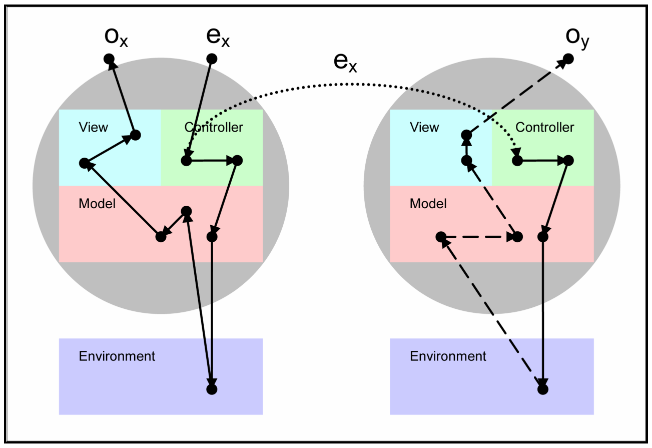

Figure 2. Externalities in Replicated Applications

In short, although MMConf, Dialogo, and the first version of Rapport succeeded in replicating the application, they failed to replicate the application's operating environment. Indeed, since they only intercepted window system events, they could only handle externalities associated with the window system (e.g., referencing a color map or a font); they could not handle externalities associated with the larger operating environment (e.g., accessing a preferences file).

4. FIXING THE PROBLEM

The key to fixing the problem is what we call *environmental replication*—replicating environmental state across the distributed applications. 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 which 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 operating environment. However, with different versions of the Java virtual machine and many core Java methods implemented using native operating systems calls, Java by itself does not solve the problem. A way to share the results of externalities is also needed. Flexible JAMM is perhaps the best such system to date [2][3].

Flexible JAMM improved upon the earlier window-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 which is implemented using native calls to the operating systems was very likely an externality. 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 3.

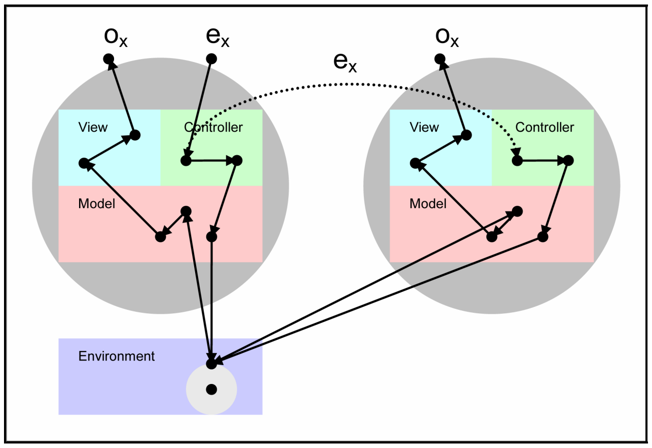

Figure 3. Flexible JAMM's Proxied Externalities

The same replicated application from Figure 1 is shown in Figure 3. Note, though, that only the environment for the moderator is shown. Flexible JAMM dynamically replaced single-user Java classes with multi-user ones. Some of the classes that were replaced were those known to cause externalities. When Flexible JAMM replaced one of these classes, it first created a small server to control access to the environmental state (shown as the small gray circle in the environment box. It then replaced the class with a small proxy which implemented the same interface. This proxy would use Java RMI to communicate with the server controlling access to the state. Since both remote collaborators and the local moderator 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 centralized server to an otherwise distributed, replicated architecture (Begole, *et al*., called this a "semi-replicated" architecture [2]). This has disadvantages of decreased fault tolerance and increased latencies. Second, care must be taken with environmental calls which can produce different results each time they are invoked (e.g., calls to the system time or to a random number generator). Third, there are additional security concerns with having multiple servers running which provide access to pieces of the operating environment. Finally, the replacement of Java objects at run time itself has several limitations (e.g., sub-classes of replaceable classes cannot, in general, be replaced).

5. THE ZIPPER APPROACH

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.

5.1 Externality Forwarding

Zipper's "externality forwarding" scheme is shown in Figure 4. Our approach, like Flexible JAMM, is to create a boundary around the application (the large gray circles). We assume that operations inside the boundary are deterministic. That is, copies of the same application in the same initial state will respond identically to the same input. Operations which cross the boundary (e.g., file system accesses) are externalities.

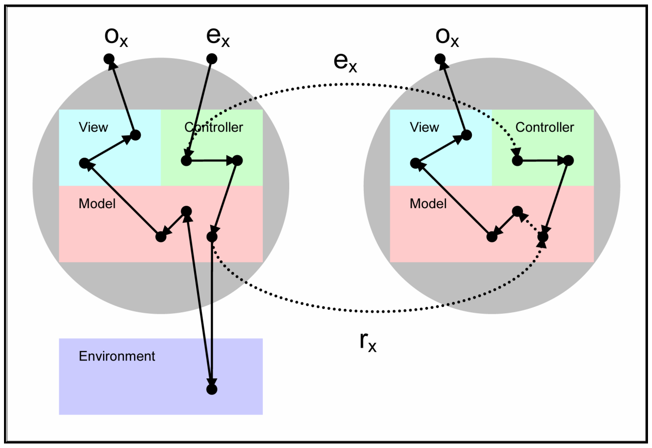

Figure 4. Zipper's Externality Forwarding

The novelty is in how we plan to handle externalities. When an externality is detected in the moderator's application, we will allow the external call to occur. Then we will package the results and send them to the other collaborators. This is shown as the dotted line labeled r_x, (for "results of an externality as a consequence of event e_x") in Figure 4. When the remote collaborator's application gets to the point of the externality, it needs to 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, o_x.

5.2 Aspect-Oriented Programming

One of the recent advances in software engineering has been the development of aspect-oriented programming (AOP) [7]. "Aspects" are special objects which 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 which 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.

Our plan is to use AOP for hooking the state-changing events of single-user applications in order to make them multi-user [4]. 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 *pointcuts* in AOP terminology), we need very little information about the applications to make them collaborative. AOP should also prove 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].

5.3 ISSUES

A number of issues with replicated application sharing are part of our ongoing research.

5.3.1 Subtleties of Externalities

Dealing with externalities is more complicated than described above. In some cases (e.g., the random number generator example) it is important to share the result of a single call and not allow multiple accesses to the environment. In other cases, it may not be necessary (or even desirable) to exactly duplicate externalities across application replicas [9][2]. Rather, the operating environment for each replica should be *effectively* the same.

5.3.2 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.

5.3.3 Latecomers

The issue of latecomers is a serious problem with replicated application sharing. In any realistic scenario, however, even starting a collaboration 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 (and perhaps only) collaborator is an example of a latecomer. We are investigating the use of state exchange via process migration to handle latecomers.

5.3.4 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 [4]. 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.

5.3.5 Finding Pointcuts

Without knowledge about an application's internals, it would be difficult to express the pointcuts to inject new behavior into the application's operation. Available source code, standards, APIs, decompilers, and tracers are very useful means of understanding the application. But some applications may be truly opaque, or too large and complicated to analyze.

6. CONCLUSION

In this note, 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 propose using aspect-oriented programming as a more standard and systematic approach for hooking events in an application. AOP should also address some limitations in earlier work that relied on dynamic class replacement. Although a number of research questions remain, we have started the Zipper project prototype in order to test the viability of our ideas and to gain a better appreciation of the problems.

7. REFERENCES

[1] Ahuja, S.R., J.R. Ensor, and D.N. Horn, "The Rapport Multimedia Conferencing System," *ACM SIGOIS Bulletin*, 9(2-3) April/July 1988, pp. 1-8.

[2] Begole, J., R.B. Smith, C.A. Struble, and C.A. Shaffer, "Resource Sharing for Replicated Synchronous Groupware," *IEEE/ACM Transactions on Networking*, 9(6) December 2001, pp. 833-843.

[3] Begole, J., M.B. Rosson, and C.A. Shaffer, "Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems," *ACM Transactions on Computer-Human Interaction*, 6(2) June 1999, pp. 95-132.

[4] Cheng, L., S.L. Rohall, J.F. Patterson, S. Ross, and S. Hupfer, "Retrofitting Collaboration into UIs using Aspects," note submitted to CSCW'04.

[5] Crowley, T., P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, "MMConf: an infrastructure for building shared multimedia applications," *Proceedings of CSCW'90*, Los Angeles, CA, pp. 329-342.

[6] Greenberg, S., "Sharing views and interactions with single-user applications," *ACM SIGOIS Bulletin*, 11(2-3) April/July 1990, pp. 227-237.

[7] Kiczales, G., *et al*., "Aspect-Oriented Programming," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 97)*, Jyväskylä, Finland, pp. 220-242.

[8] Lauwers, J.C. and K.A. Lantz, "Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems," *Proceedings of CHI'90*, Seattle, WA, pp. 303-311.

[9] Lauwers, J.C., T.A. Joseph, K.A. Lantz, A.L. Romanow, "Replicated architectures for shared window systems: a critique," *ACM SIGOIS Bulletin*, 11(2-3) April/July 1990, pp. 249-260.

[10] NetMeeting: http://www.microsoft.com/netmeeting.

[11] Sametime: http://www.lotus.com/sametime.