Resource Sharing for Replicated Synchronous Groupware

James "Bo" Begole, Randall B. Smith,
Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303-4900
{Bo.Begole,Randall.Smith}@Sun.Com

Craig A. Struble, Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
{cstruble,shaffer}@cs.vt.edu

Abstract

We describe problems associated with accessing data resources external to the application, which we term *externalities*, in replicated synchronous collaborative applications (e.g., a multiuser text editor). Accessing externalities such as files, databases, network connections, environment variables and the system clock is not as straightforward in replicated collaborative software as in single-user applications and centralized collaborative systems. We describe *ad hoc* solutions that have been used previously. Our primary objection to the *ad hoc* solutions is that the developer must program different behavior into the different replicas of a multi-user application, which increases the cost and complexity of development.

We introduce a novel general approach to accessing externalities uniformly in a replicated collaborative system. The approach uses a semi-replicated architecture where the actual externality resides at a single location and is accessed via replicated **proxies**. The proxies multiplex input to and output from the single instance of the externality. This approach facilitates the creation of replicated synchronous groupware in two ways: (1) developers use the same mechanisms as in traditional single-user applications (2) developers program all replicas to execute the same behavior. We describe a general design for proxied access to read–only, write–only and read–write externalities. We discuss the tradeoffs of this semi-replicated approach over full, literal replication and the class of applications to which this approach can be successfully applied. We also describe details of a prototype implementation of this approach within a replicated collaboration-transparency system, called Flexible JAMM (Java Applets Made Multi-user).

KEYWORDS: computer-supported cooperative work, collaboration transparency, distributed file systems, Flexible JAMM, groupware, distribution architectures, Java^{TM} platform, input–output, object-oriented systems

1 Introduction

With today's proliferation of computers and near universal networking, the trend toward *personal computing* has evolved to *inter*-personal computing. People collaborate continually in their physical environment but, despite the increasing tendency for work to involve a computer, there is little support for synchronous collaboration in today's systems.

A number of factors contribute to this deficiency, perhaps primary among them is the cost of including support for synchronous collaboration in an application. Several technical and human factors must be addressed that are not necessarily required in a single-user application, such as system distribution, concurrency control and collaborative usability (Grudin, 1994; Patterson, 1991). A key technical issue is the sharing of external system resources, such as files, sockets, and the system clock. We call such resources **externalities** because they represent state necessarily external to the application. Groupware toolkits (Burridge, 2000; Chabert *et al.*, 1998; Dourish, 1998; Graham *et al.*, 1996; Roseman & Greenberg, 1996; Lee *et al.*, 1996), which facilitate the creation of synchronous multi-user software, do not address access to externalities.

This paper presents common problems related to sharing externalities in real-time collaborative applications that use a replicated architecture. In Section 2, we describe the range of groupware architectures,

[0]Sun, Sun Microsystems, Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All other products mentioned herein are trademarks or registered trademarks of their respective owners.

their tradeoffs, and the issues related to sharing externalities under replicated systems. In Section 3, we describe approaches to handling externalities. *Ad hoc* solutions that have been discussed in the literature are described in Section 3.1. The primary problem with these *ad hoc* approaches is that the application developer must program the replicas to behave differently depending on each's role. This complexity adds to the cost of developing collaborative applications over traditional single-user applications. Section 3.2 describes general solutions to handling externalities and introduces a novel semi-replicated approach in which the actual externality is accessed via replicated **proxies**. This approach allows application developers to program the resource management portions of all replicas *uniformly.* In Section 4, we describe a prototype implementation of this approach as part of a replicated collaboration-transparency system, called Flexible JAMM. To our knowledge, this is the first time that the range of problems related to the issue of accessing externalities in synchronous replicated collaborative applications has been addressed explicitly or that the semi-replicated proxy solution has been presented.

2 Collaborative Application Architectures

Synchronous collaborative systems are inherently distributed. That is, components of the system execute on different machines and communicate via a network. Distributed software architectures fall in a range from **centralized**, where all of the shared data are maintained and processed at a single location, to **replicated**, where each site maintains and processes a complete copy of the shared data (Lantz, 1986; Coulouris *et al.*, 1994). The diagrams in Figures 1 and 2 illustrate the key components and communication paths between processes of a conceptual two-user collaborative system under fully centralized and replicated architectures.

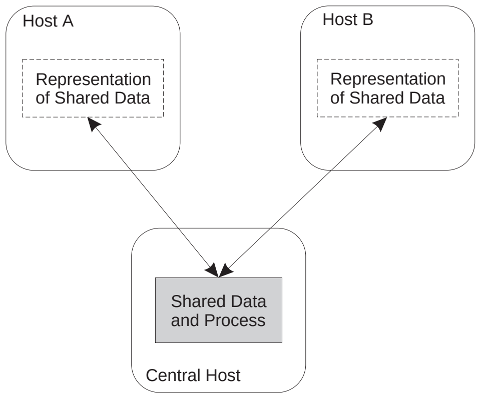

Figure 1: Centralized architecture. The shared data, indicated by the shaded box, exist and are processed at a single host. A person on a remote host views and manipulates the data via a representation, indicated by a dashed box. Here, two hosts, A and B, have access to the centralized data.

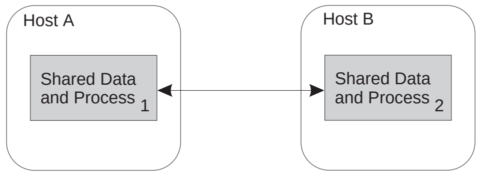

Figure 2: Replicated architecture. Each host contains and processes a full copy of the shared data. When the data change on one host, all replicas must be made consistent. Arrows indicate network traffic.

2.1 Architecture Tradeoffs for Synchronous Groupware

Centralized architectures guarantee consistency of shared data because there is only one copy. On the other hand, a centralized implementation typically requires higher network bandwidth to distribute display information than does a replicated implementation which can distribute only minimal update information. Fully centralized systems also impose strict What You See Is What I See (WYSIWIS), where the participants see exactly the same view of the shared application at the same time (Stefik *et al.*, 1987), which disallows independent work. Furthermore, centralized implementations are less responsive to user input due to round-trip latency as each user interaction must travel to and from the central location. Finally, centralized approaches are potentially less fault tolerant than replicated because the central host is a single point of possible system-wide failure.

Replicated and semi-replicated architectures can have lower bandwidth requirements and support important collaborative usability principles. First, a replicated system can provide faster response to user input as the local copy can be updated before remote copies. Additionally, the constraint of strict WYSIWIS can be relaxed by having a different view of the shared data at each replica. Independent simultaneous work is further supported under replication by allowing participants to modify their local copy of data and merging the changes with remote copies using techniques such as operational transformation (Sun & Ellis, 1998). However, maintaining consistency among shared data replicas is more complex than sharing a single copy of centralized data. Despite the increased complexity, groupware toolkits and applications tend to favor replicated and hybrid architectures (Begole, 1998; Greenberg & Roseman, 1999).

One class of real-time groupware that generally does not use replicated architectures is application-sharing systems, which provide the shared use of existing single-user applications. All currently available commercial application-sharing systems (e.g., Microsoft NetMeeting and SunForum) use centralized architectures as do most research application-sharing systems. Such systems are useful for tightly-coupled collaborations where the collaborators work closely together. However, they have been found to use network resources inefficiently and to be too limiting for collaborations where group members work with any degree of independence because they lack support for fundamental groupware principles: concurrent work, relaxed WYSIWIS, and group awareness (Begole *et al.*, 1999; Prakash & Shim, 1994; Reinhard *et al.*, 1994; Schuckmann *et al.*, 1996).

An important observation is that a *purely* centralized architecture is not possible in practice because a representation of the shared data must be replicated to each of the participating client machines. At the very least, a graphic representation of the shared data will be replicated. Therefore, all synchronous collaborative systems are in fact semi- to fully replicated. The practical question is to decide at which layer should replication occur: screen pixels, user interface data, user interface behavior, in-memory application data, or externalities. Replication at each layer has tradeoffs, as discussed by Dewan (1999).

To summarize, this section has described the tradeoffs when comparing the extreme ends of the range of conceptually centralized to fully replicated architectures. Centralization guarantees that all participants access the same shared data. However, as a system's architecture approaches complete centralization it constrains usability and typically requires higher network bandwidth than architectures tending toward replication. Although most application-sharing systems use a highly centralized approach, replicated architectures are favored by groupware toolkits and *ad hoc* groupware applications primarily because replication allows more efficient use of the network and advanced user-level support for collaboration as coworkers shift between tightly and loosely coupled concurrent work.

2.2 Problems with Sharing System Resources

In addition to the user's keyboard, mouse, and screen, there are many other sources of input and destinations for output: printers, files, databases, network connections, other processes, etc. There is also an application's runtime environment which provides inputs such as the current time and the values of environment variables. We use the term **externality** for a source of input or output that is external to the application other than user input and display output. We exclude user input–output from the definition because the problems of handling them are fundamentally different than those of other input–output resources, in many ways the problems are reversed. Inputs generated by multiple users must be merged in some fashion, often to be delivered to the application as a single stream of user-generated input. Conversely, input read from a single externality is multicast to multiple replicas. Display output *must* be replicated to each user. Conversely, output destined for an externality cannot always be replicated, as we will discuss shortly. Solutions to issues surrounding user input–output can be found in the literature regarding groupware toolkits and applications (Begole *et al.*, 1999; Greenberg & Roseman, 1999; Sun & Ellis, 1998), whereas externality input–output in groupware is not addressed elsewhere.

In general, there is no problem sharing an externality in a centralized system because only the single central process is accessing the externality. In contrast, copies of a replicated system generally cannot be permitted to access an externality directly because not all replicas may have access to the externality or the value of the externality at each replica may not be the same. For example, the system clock on each host will return a different value. As another example, if a replicated application needs to read a file, on which host should the file be opened? Suppose a file of the same name resides on each host but contains different data. In these examples, it is possible for replicas to receive different input and this can result in inconsistent states. If we assume the replicas are copies of the same deterministic process, we can only guarantee their consistency when all replicas receive the same input. Ensuring consistency among the replicas requires that they run in *effectively* the same environment. Techniques to provide the illusion that replicas share one environment and therefore receive the same input are described in the next section.

Not all externalities should necessarily provide the same data to all replicas. For example, applications may use "environment variables," such as the user's home directory, current working directory, and command path. A replica may behave incorrectly if it is given the value of a variable from another replica's environment. For example, on UNIX systems, the user's home directory is stored in an environment variable, named `$HOME,` and is different for each user. If replica A is running on one user's machine and requests the value of `$HOME` and is given the home directory of the user running replica B on a different machine, replica A would fail to access that directory. Therefore, the developer must take care to selectively distribute only those parts of the external environment required to maintain consistency among the replicas.

Replicated output can also pose a problem. In some cases output operations may be considered idempotent. That is, it can be acceptable to allow each replica to generate output redundantly. In other cases, however, generating the same output multiple times is not desired. For example, other than being redundant, it would be acceptable for each replica of a collaborative editor to write a separate copy of a file on its local host. However, it would be annoying if each replica of a collaborative email composer sent a copy of an email message to the same recipient. The developer must consider these possibilities and ensure proper behavior in a replicated collaborative application.

3 Handling Externalities in Replicated Systems

Externalities are trivially handled under centralized architectures but are more difficult under the replicated architectures favored by groupware toolkits and groupware applications. Current groupware toolkits (surveyed in (Begole, 1998)) provide no abstractions to facilitate replicated input to, or output from, externalities. A developer using one of these toolkits must be aware of the issues and provide solutions to manage externalities correctly. Generally, such *ad hoc* approaches require replicas to access externalities in a non-uniform way, leaving it to the developer to coordinate access among the replicas. In contrast, two general approaches allow uniform access to externalities and remove coordination concerns from the application developer: (1) full environment replication and (2) semi-replicated proxies. This section describes the tradeoffs of the *ad hoc* and general solutions.

3.1 *Ad Hoc* Solutions

In some cases, it is possible for all replicas to access a single instance of the externality. One example is a file that is referenced by a Uniform Resource Locator (URL) and delivered by a web server which can be loaded by each replica independently. Replicas of a multi-user whiteboard application, for example, could use this approach to load clip-art image files from the World Wide Web (WWW).

Often, though, an externality is only directly accessible from one replica. For example, a particular file may reside on only one of the hosts in the replicated system. In the case of the system time, all hosts *do* have access to a local clock, but the states of those clocks differ. Therefore, we may designate one host as the source of that externality to ensure that all replicas receive the same data in response to the same query. The replica that accesses the source may be referred to as the "master," and the other replicas may be called "slaves." Figure 3 illustrates this approach.

3.1.1 Explicit Distribution

Consider a text editor that reads an input file and appends the file contents to an in-memory document. There are several ways a replicated multi-user editor might handle this situation. Figure 4 shows a pseudo-

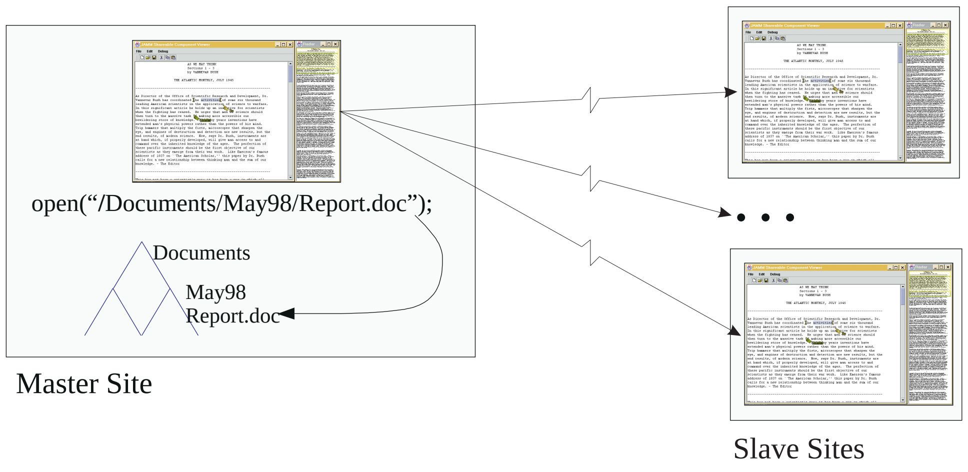

Figure 3: Illustration of a shared editor application that explicitly distributes externality data among replicas. The "master" replica on the left accesses a file and distributes the contents to the "slave" replicas on the right.

code fragment in which data are read by the master replica. The master replica applies the data to its local copy of the shared document, then explicitly generates a message containing the data and sends the message to all other replicas. A facility to send a message to all replicas other than the originating replica is available in many groupware toolkits (e.g., GroupKit (Roseman & Greenberg, 1996), and the Java Shared Data Toolkit (Burridge, 2000)) and is easily implemented in *ad hoc* groupware applications.

3.1.2 Implicit Distribution

In the above examples, the developer was required to specify a message protocol, which consists of several steps: the sender creates a message containing the information to distribute, the sender explicitly sends the message, and each recipient must parse and handle the message. Each type of message must have a unique identifier, so that the recipient can handle each type differently. There must be a unique message type for each operation that a replica can perform upon receipt of a message. For example, in Figure 4, the program performs one action (`appendLocal()`) upon receipt of an `appendText` message, and performs another action upon receipt of a `someOtherMessage` message.

Rather than creating a message protocol explicitly, it is possible to invoke behavior on all replicas remotely using a remote procedure call (RPC) or remote object method invocation provided by distributed object technologies, such as the Common Object Request Broker Architecture (CORBA) (Mowbray & Zahavi, 1995), Microsoft Distributed Component Object Model (DCOM) (Sessions, 1997), and Java™ Remote Method Invocation (RMI) (Wollrath *et al.*, 1996). A groupware extension to RPC is provided by GroupKit (Roseman & Greenberg, 1996), called Multicast Remote Procedure Call (MRPC), which adds the capability to make the invocation on *multiple* remote processes simultaneously. Using MRPC, the ten lines of pseudo-code in Figure 4 could be simplified to the six lines seen in Figure 5.

MRPC is conceptually simpler to program than creating and handling a message protocol explicitly. Nevertheless, although MRPC mitigates the tedium of creating a message protocol, the complexity of coordinating access between master and slave replicas remains. Only the master replica should invoke the `readFile()` method. MRPC and other remote invocation mechanisms solve only half the problem.

3.2 General Solutions

The primary disadvantage of the preceding approaches is that the developer must program different behavior into the "master" and "slave" replicas. This complexity contributes to the higher cost of developing a multi-user application than that of an otherwise equivalent single-user application. There are general solutions to

```
 1.  procedure readFile(inFile) {
 2.    while (inFile is not empty) {
 3.      read data from inFile and store in a buffer
 4.      appendLocal(buffer);
 5.      sendMessageToOthers("appendText", bufferSize, buffer);
       }
     }
 6.  procedure receiveMessage(msgType, netInput) {
 7.    if (msgType equals "appendText") {
 8.      read data from netInput and store in inputChars
 9.      appendLocal(inputChars);
       } else if (msgType equals "someOtherMessage") {
         // do something else ....
       }
     }
10.  procedure appendLocal(inputChars) {
11.    append inputChars to in-memory document
     }
```

Figure 4: Sample pseudo-code to read a file into a replicated collaborative text editor. Data are read from the file, appended locally, and then sent in a message to all application replicas (lines 1–5). When the message is received by each replica, `receiveMessage()` is invoked (line 6). When the message type is "`appendText`," the text is extracted from the message and then appended to the local copy of the document by invoking `appendLocal()` (lines 9 and 10). Lines 1–5 and 10–11 are invoked only by the master, 6–11 by all slave replicas.

```
 1.  procedure readFile(inFile) {
 2.    while (inFile is not empty) {
 3.      read data from inFile and store in a buffer
 4.      invokeOnAll("appendLocal", buffer);
       }
     }
 5.  procedure appendLocal(inputChars) {
 6.    append inputChars to in-memory document
     }
```

Figure 5: Sample pseudo-code using a multicast remote procedure call to directly invoke the procedure that appends data to the document on all replicas. Lines 1–4 are invoked only by the master, 5–6 by all replicas (master and slave).

externality access, however, that allow the developer to define the same behavior in all replicas. Furthermore, it is possible to use the same mechanisms as those of accessing input–output resources in a single-user, non-distributed application.

Section 3.2.1 describes a straightforward approach to handling externalities: full replication. Although full replication can be effectively applied for files, it cannot be applied to all externalities and is therefore not a complete solution. In Section 3.2.2, we introduce a complete, general solution to handling externalities in a groupware application based on the use of replicated proxies to a single instance of a shared externality.

3.2.1 Full Externality Replication

One approach to externality distribution is to completely replicate the externality so that each replica has individual access to an identical copy. This approach was used to share files in MMConf (Crowley *et al.*, 1990), a replicated groupware toolkit, and Dialogo (Lauwers, 1990), a replicated collaboration-transparency system. In Dialogo, a directory was designated as the "conference directory," and any file placed in it was automatically copied to other participants' conference directories. The users of shared applications in these systems confined their access to files in the conference directory.

Although literal copying can be effective for shared files, there are still some difficulties related to uniform file access and literal copying does not work for many other types of externalities. One problem arises when the collaborative application uses the fully qualified path to access a file. If each participant's conference directory resides in a different absolute path, some replicas may fail to locate the file. Additionally, differing file naming conventions (e.g., Macintosh versus UNIX file systems) prevent uniform access to files across replicas running on heterogeneous systems. Additionally, literal replication does not help in cases where the externality will return a different value depending on the machine on which it exists, such as environment variables (e.g., host name) and the system clock. Finally, in some cases it is infeasible or impossible to literally replicate an externality, such as the network connection to an exclusive service.

One advantage of literal replication is that it allows a user to continue working in case of a network failure. In some cases, however, the advantage of being able to continue working independently offline may be offset by the requirement to merge conflicting edits later. In any case, from the perspective of the isolated collaborator, the synchronous collaboration is broken.

3.2.2 Proxied Externalities

It can be impractical to make a literal copy of each externality, but it is still possible to provide uniform access to an externality from all replicas. We now introduce a semi-replicated approach where the externality resides physically at a single location and is accessed via replicated **proxies** (Gamma *et al.*, 1996) that multiplex input to and output from the actual externality. To the application, the proxy acts in place of the actual externality and the programmer accesses the proxy with the same code that would be used to access the externality itself.

Figure 6 shows the design for an object-oriented proxied externality which consists of a client, called the externality proxy, and an externality server. The proxy implements the interface of the original externality so that the proxy will pass the same runtime type checks as the original. The server holds a reference to an actual externality from which the server acquires or writes data.

Proxies and servers behave differently depending on whether the externality only provides input to the application, only accepts output from the application or both provides input and accepts output. We describe the algorithms followed for each type next.

Input-only externalities, such as the system clock or read-only files, are handled in the following manner. When a proxy is created, it registers with the corresponding externality server which assigns a unique identifier to the proxy, so that each proxy's request can be tracked. When the application replica reads from the proxy, the proxy increments a request counter by one and sends the request number and unique proxy identifier along with the other request parameters to the externality server. Upon receipt of a request, the server checks the request number to see if it is higher than any request number it has serviced previously. If so, then this is the first proxy to make this request. The real externality is accessed and the data are returned to the requesting replica. The server caches the data in a table and maps that request number with that data. As each replica makes the same numbered request, the server returns the data for

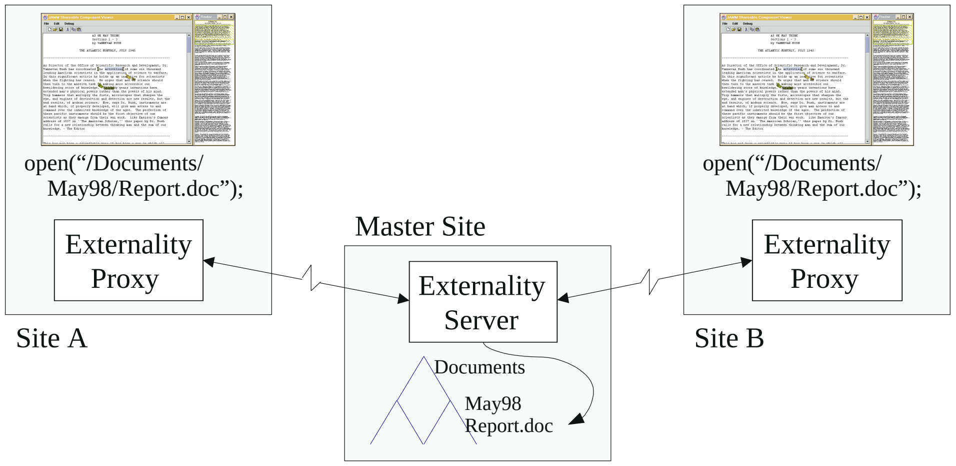

Figure 6: Shared editor replicas access the externality uniformly via proxies. The externality resides physically on a central site and is accessed by proxies at each replica.

that request number. Data are only cached as long as they are needed. When all of the active replicas have made the same request, the cache space for that request is released. Figure 7 contains pseudo-code summarizing how an input-only externality server handles read requests.

To improve the speed at which other proxies receive the value, the server may send the result immediately after the first proxy makes that request, as described by Patterson *et al.* (1996) and Strom *et al.* (1998). If a collaborator leaves the session deliberately, that replica's proxy will notify the server of its impending disconnection so that the server can discontinue tracking its requests and caching data for it. Additionally, if a replica is separated from the session due to a network fault, the server detects the disconnection of the proxy through the absence of a heartbeat signal sent periodically from each proxy. Other fault detection mechanisms, such as renewable leases, can be used.

```
LocalReadOnlyExternality realExtern;

public RetType readRequest(ProxyID proxy, int requestNumber,
                           Type1 param1, Type2 param2, ...) {
    RetType retval;

    if (proxy has the highest requestNumber) {
        retval = realExtern.readRequest(param1, param2, ...);
        store request, return value, and parameters in cache table
            keyed by the requestNumber;
    } else {
        access cache table by requestNumber
        retval = stored return value;
        add proxy to list of proxies that have made this request;
    }
    if (proxy is the last replica to make the request) {
        remove the request's cached data from the table
    }
    return retval;
}
```

Figure 7: Server-side pseudo-code for a read request of a proxied input-only externality.

Output-only externalities that do not return a value from a write request, such as write-only files and output streams in C++ and Java, are handled as follows. One proxy is designated as the "master" and only

```
RemoteOutputOnlyExternalityServer remoteOutputOnlyServer;

public void writeRequest(Type1 param1, Type2 param2, ...) {

    increment requestNumber;

    if (this is the master proxy) {
        remoteOutputOnlyServer.writeRequest(requestNumber, param1, param2, ...);
    } else {
        store request and parameters in master-recovery cache;
    }
}
```

Figure 8: Proxy-side pseudo-code for a write request of a proxied output-only externality. In case the master is lost, any of the remaining proxies can take over as master, determined by a distributed consensus algorithm, by re-applying the write requests stored in its master-recovery cache. The cache is periodically flushed (not shown here).

its write requests are actually sent to the externality server and written to the externality. The designation of the master proxy can be arbitrary but should be a proxy that uses network bandwidth most efficiently (i.e., the "closest" proxy in network terms). All write requests made by other replicas are not sent to the server. However, all proxies store the write requests so that each is able to take over as the master in case of a fault at the master. In case of such a fault, a distributed consensus algorithm may designate a new master that then applies the write operations that occurred since the fault. To allow proxies to flush unneeded data, the server periodically sends a notice to all proxies of the last applied write operation. The pseudo-code in Figure 8 summarizes how an output-only externality proxy handles write requests.

Input-and-output externalities, such as read-write files or databases, are handled by combining the above two approaches, summarized by the pseudo-code for an input-and-output externality server in Figure 9. Again, one proxy is designated as the "master" and only its write requests are sent to and applied to the actual externality. To ensure correctness, it is necessary to synchronize the proxies at the point of each read that follows a write request. Otherwise, it would be possible for a fast-running slave proxy, whose write requests are dropped, to read a value incorrectly before the master writes an update to it. The incorrect value would then be cached and returned to other proxies, including the master, when each replica made the corresponding read request. To prevent such incorrect results, it is necessary for proxies to be synchronized with the master following writes. It is sufficient to synchronize the proxies prior to the read following one or more writes, rather than after each write, because there is no risk of inconsistency until a read is performed. This saves the proxies from having to send a synchronization-check message to the externality server at the point of each write request.

Synchronization is performed in the following way. Recall that each proxy increments its request number by one with each read and write request. When a non-master proxy makes a read request following one or more dropped writes, the read's request number will be more than one greater the proxy's previously sent request number because the requests for the preceding writes were not sent. To detect this, when the externality server receives a request, the server checks the difference between this request number and that proxy's previous request number. If the difference is greater than one, the server needs to synchronize this proxy with the master before returning the value of the read. The externality server synchronizes the proxy by blocking the proxy's request until its request number is less than or equal to one more than the master's last request number. When that condition is true, the master has completed the write request that precedes the read issued by the proxy. Therefore, the proxy can safely read the data. The server will read the actual externality, return the value to the proxy and cache it. Corresponding read requests from other proxies, including the master, will be given the cached value. Subsequent reads from that proxy, up to the next write, do not need to wait for the master.

```
LocalReadAndWriteExternality realReadWriteExternality;

public RetType readRequest(ProxyID proxy, int requestNumber,
                           Type1 param1, Type2 param2, ...) {
    RetType retval;

    if ((requestNumber - last requestNumber for proxy) > 1) {
        // synchronize proxy with master
        while ((requestNumber - master's last requestNumber) > 1) {
            block this proxy until master makes another request
        }
        access cache table by requestNumber
        retval = stored return value;
        add proxy to list of proxies that have made this request;
    } else if (proxy has the highest requestNumber) {
        retval = realExtern.readRequest(param1, param2, ...);
        store request, return value, and parameters in cache table
            keyed by requestNumber;
    } else {
        access cache table by requestNumber
        retval = stored return value;
        add proxy to list of proxies that have made this request;
    }
    return retval;
}
```

Figure 9: Server-side pseudo-code for a read request of a proxied input-and-output externality. Note that non-master proxies set the checkSynch value to true only for read requests that follow a write. Write requests and fault tolerance are handled as in Figure 8.

As an example, consider an externality with only two operations: `void setValue(int newValue)`, which sets the value of the externality; and `int getValue()`, which returns the value of the externality. Suppose each replica will execute the following series of operations on the externality.

```
setValue(5);
x = getValue();
setValue(x+1);
y = getValue();
```

On all replicas, the result should be `x == 5` and `y == 6`. Table 1 traces how the server responds to two proxies issuing this series of read and write requests.

3.2.3 Applicability and Limitations

This approach of proxied externalities is applicable to systems in which the replicas access the externality using the same requests in the same order. Thus, it is particularly suited to replicated collaboration-transparency systems, such as Dialogo (Lauwers, 1990) and Flexible JAMM (Begole *et al.*, 1999), where identical copies of the shared application are executed on each collaborator's host. Each replica makes the same requests in the same order because each replica is a copy of the same deterministic process.

Proxied externalities are also well suited to applications specifically designed to be used collaboratively so long as the replicas make the same calls to the externalities in the same order. The replicas in such a system are not required to behave uniformly in any other respect. Generally, replicas in a groupware system differ primarily in their views of shared data, not in how they acquire or store the data.

Because this semi-replicated system has a centralized component (the actual externality) it carries two disadvantages common to centralized architectures. The first is that proxied externalities are less fault tolerant than full literal replication (Section 3.2.1). Under the proxied approach, a user cannot continue

Time	**Req. Num**	**Proxy A (master)**	**Req. Num**	**Proxy B**	**Value**
1	1	`setValue(5)` (This is sent and executed because A is the master.)			5
2	2	`x = getValue()` (5 is returned and is cached, associated with req # 2.)			5
3			1	`setValue(5)` (This is not sent because B is not the master.)	5
4			2	`x = getValue()` (req # 2 is in the cache so 5 is returned. This is the last expected appearance of req # 2, so its cached value is cleared.)	5
5			3	`setValue(x+1)` (This is not sent because B is not the master.)	5
6			4	`y = getValue()` (Req # 4 is 2 greater than B's last req # and 2 greater than the master's last req #. So, the server blocks on this request.)	5
7	3	`setValue(x+1)` (This is sent and executed as A is the master.)			6
8			4	(Now req # 4 is only 1 greater than the master's last req #. So, the server returns 6 and caches it associated with req # 4.)	6
9	4	`y = getValue()` (Req # 4 is in the cache so 6 is returned. This is the last expected appearance of req # 4, so its cached value is cleared.)			6
Result	`x==5, y==6`		`x==5, y==6`		6

Table 1: A series of read and write requests sent from two proxies to an input-and-output externality server. Proxy A is the "master" and Proxy B is a "non-master."

```
1.  procedure readFile(inFile) {
2.      while (inFile is not empty) {
3.          read data from inFile and store in a buffer
4.          append buffer to in-memory document
        }
    }
```

Figure 10: Sample pseudo-code using a proxied input file. All replica's use the same behavior, relieving the developer from managing the role of each replica. An additional benefit is that the developer can use code similar to that used to access externalities locally in a traditional single-user application.

to work offline in case of a network fault because the actual externality is not available locally. We note, however, that the ability to work alone would disrupt the nature of a synchronous collaboration in any case. Another issue related to network faults is that the centralized externality is a single point of possible failure. No replica can continue if the externality server is unreachable. Generally, although not a requirement, the externality server would reside on the same host as one of the replicas and at least that replica could continue, although clearly the collaboration would be broken.

The second disadvantage is that the speed of data retrieval is dependent on network latency as each request must travel from the proxy to the server and return. This could result in unacceptable performance in systems that frequently query an externality such as the system time. An additional performance limitation is seen in the case of input-and-output externalities where proxies are synchronized with the master at the point of a read following one or more writes. If the master is more sluggish than the other proxies, this synchronization step will prevent the other replicas from executing as quickly as they could otherwise. However, when the master runs at speeds comparable to or faster than the other replicas and the master and externality server are co-located, the synchronization delay is minimal. There is no synchronization delay imposed on input-only and output-only externalities. Note that network latency only affects the manipulation of data *external* to the application. Access to externalities is relatively infrequent when compared to the processing and network traffic involved in handling user inputs during a collaboration.

3.2.4 Benefits

This semi-replicated access to externalities allows collaborative systems to use a replicated architecture, which has network and usability advantages, while providing shared access to externalities that are not fully replicable. This capability provides benefits for application-sharing systems and collaboration-aware applications (applications designed to be used collaboratively).

Current commercial application-sharing systems use centralized architectures in part because these prevent the possibility of inconsistent shared data which can arise from problems accessing externalities, as we discussed in Section 2.2. However, centralized application-sharing systems have been shown to use network resources inefficiently and impose an inflexible style of collaboration by not adequately supporting key groupware principles: concurrent work, relaxed WYSIWIS, and detailed group awareness (Begole *et al.*, 1999). Proxied externalities make replicated architectures more viable for application-sharing systems which can alleviate the usability problems found in conventional, centralized systems.

This approach to handling externalities is also beneficial to the development of collaboration-aware applications. First, externalities may be accessed using code similar to that used in a single-user application (with additions for handling network faults). Second, at the application level, all replicas have the same behavior because the master–slave coordination is pushed into the proxies. This simplifies collaborative application development leading to faster development and more reliable code. As an illustrative example, consider that whereas the *ad hoc* approaches described in Section 3.1 used minimally two methods consisting of six lines of pseudo-code (not counting master/slave determination and management), as seen in Figure 5, the proxied approach can use one method consisting of the four lines shown in Figure 10.

How much savings there would be in lines of real code, as opposed to pseudo-code, between *ad hoc* and proxied implementations depends on the language and development environment but clearly there is a savings in terms of reduced complexity at the application level. If one were to augment a modern office productivity suite, such as OpenOffice (OpenOffice.org, 2001), with collaborative capabilities, such savings would be substantial. To handle externalities in such a system would first require the development of a package to handle proxying the externalities, such as the prototype implementation we describe in the next section. Beyond that, at the application level, the proxied externality approach would require nearly the same amount of code as in the current implementation whereas an *ad hoc* approach would require additional code for master/slave coordination and for file content propagation.

4 A Proxied Externality Prototype

We have implemented a prototype of our proxied externality approach as part of a replicated application-sharing system for the Java$^{\text{TM}}$ platform, called Flexible JAMM (Java Applets Made Multi-user) (Begole *et al.* 1997; 1998; 1999). To maintain transparency when replacing an externality with a proxy, we modified core library classes and native platform code in the standard Java 1.1.6 runtime environment. As a result, the Flexible JAMM implementation of proxied externalities uses a non-standard Java runtime environment.

Figure 11 shows our class design in the proxy and server implementation for a Java read-only file resource, called `java.io.FileInputStream`. The proxy implements a `Proxy` interface which defines a method for connecting to the server and registering this proxy with a unique identifier (`connectToMaster()`). This method is called when the application replica creates the proxy object to register it with the server. In this way, the server can keep track of each proxy's requests and can release data after all proxies have made the request for that data. The `ProxyFileInputStream` class contains a locator, `remoteResourceLocator`, which is the address of the externality server and contains the address of a registry and a unique identifier for the externality server of that proxy. The `ProxyFileInputStream` also contains a reference to an interface for the remote externality server, `RemoteFileInputStream`, which is implemented by `RemoteFileInputStreamImpl`. The `RemoteFileInputStream` implements `RemoteExternality`, which defines a method by which proxies register themselves (`registerProxy()`), and a method that proxies can use to find out their unique connection number (`getConnectionNumber()`). `RemoteExternality` implements `java.rmi.Remote`, which is required of all Java RMI remote objects.

In addition to implementing a proxy and server class for each externality class, the original class is modified to contain a reference to either an externality proxy or a local externality (e.g., see the `ProxyFile-InputStream` and `LocalFileInputStream` fields of `java.io.FileInputStream` in Figure 11). This approach follows the **bridge** design pattern, described by Gamma *et al.* (1996). A bridge decouples an abstraction (e.g., `FileInputStream`) from its implementation (e.g., a physically remote file or a physically local file), allowing the implementation to change at run time.

This extension of the design described in Section 3.2.2 allows the proxy to be used in an application in both single- and multi-user mode. When the proxy is used in a single-user application, the object accesses the local externality directly. If the application is later shared, the object switches from accessing the local externality directly to accessing a remote externality server. The original local externality is wrapped by the externality server. In this way, a single-user application can switch to multi-user access dynamically. Figure 12 shows (a) an example of a single-user application with a `FileInputStream` object, and (b) the introduction of proxies after the application has been shared.

Although we were able to proxy file resources transparently in Flexible JAMM, we encountered a problem with proxying the system clock (`java.lang.System.currentTimeMillis()`). The problem is that the system clock is not only accessed by the application within the Java virtual machine (VM) but also by the VM itself. Consistent application replica state does not depend on these VM-level calls sharing the same global time. Therefore, to maintain efficient VM performance, we did not simply replace the reference to the system clock with a proxy. For each access of the system clock, we checked to see if the request came from an application-level object. If so, the proxy was accessed, otherwise the local machine was accessed. Therefore, the implementation of `java.lang.System` is more complex than the other externality classes in that it performs an additional check before accessing the data.

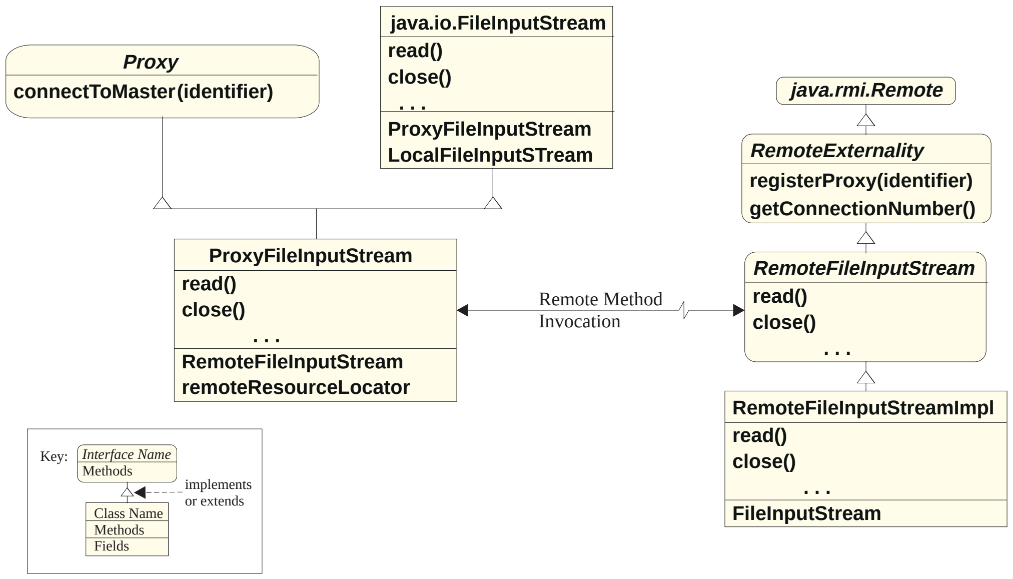

Figure 11: Class diagram for prototype implementation of a proxy (left) and server (right) for a Java platform file externality, called `FileInputStream`.

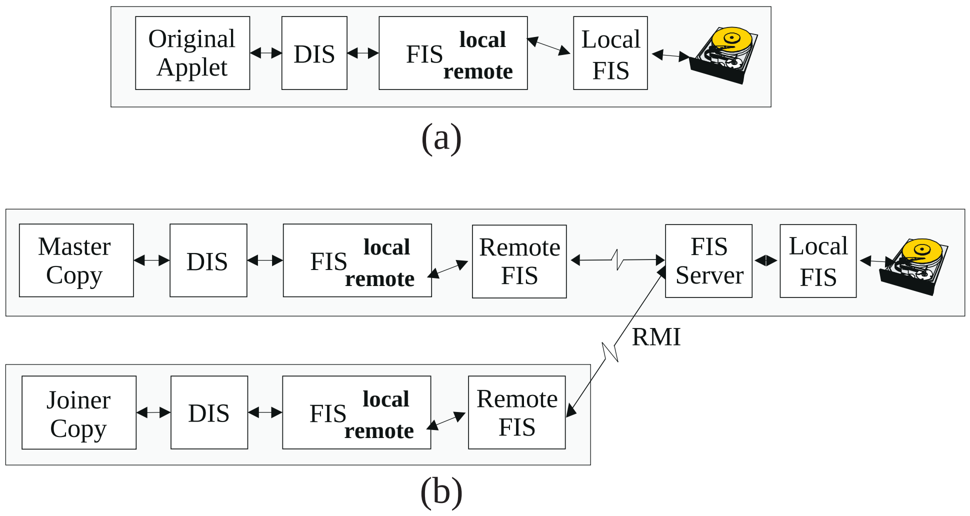

Figure 12: (a) The original applet has a `java.io.DataInputStream` (DIS) connected to a `java.io.FileInputStream` (FIS), which refers to a Local FIS, which in turn reads a file from a physical disk. (b) A proxied FIS object is shared by two application replicas. The original Local FIS resides at the original (master) host and is accessed via a server object. Each replica has a FIS Proxy which uses RMI to access the original externality at the server.

Externalities	**Rationale**
java.io.FileInputStream	FileInputStream reads a file sequentially.
java.io.FileOutputStream	FileOutputStream writes to a file sequentially.
java.io.RandomAccessFile	A RandomAccessFile reads from or writes to a file in a nonsequential manner.
java.io.File	File contains platform-specific information for path and file separators, and provides operations to create a directory, determine a file's access permissions and type, etc.
java.io.FileDescriptor	FileDescriptor represents an open file or socket. Applications should not create FileDescriptor objects directly, so this externality should not need to be proxied.
java.lang.Runtime	Runtime provides information about the runtime environment, such as the amount of free memory. Runtime should not be proxied, because each replica will have a machine-specific runtime environment.
java.lang.Process	Process objects are returned by exec() calls on Runtime and provide access to a child process's input and output streams. Process is an abstract class and is therefore not inherently an externality, but subclasses of it are.
java.lang.System	System accesses properties, standard input and output streams (stdin, stdout, and stderr) and the current time.
java.util.Random	Random generates pseudo-random numbers based on an initial seed value. The system time is the default seed. Therefore, because the system time will be proxied, Random does not need to be treated as an externality.
java.net.InetAddress	InetAddress provides internet address information including the local host name and address. InetAddress itself has no native methods, but contains a platform-specific subclass, InetAddressImpl, which has native methods.
java.net.DatagramSocket java.net.Socket	These classes provide input and output to a network. These classes do not contain native methods themselves, but use platform-specific subclasses of the abstract SocketImpl and DatagramSocketImpl classes.

Table 2: Externalities in version 1.1 of the Java class library.

4.1 Java Externality Classes

Identifying externality classes in Java is straightforward. As a rule, a Java externality class will access the actual externality via a **native** method, which is a platform-specific implementation of a method to which the platform-independent Java virtual machine passes control. For example, the `FileInputStream` reads an integer from a physical file via a native method named `read()`. Externality classes in version 1.1 of the core Java class library are listed in Table 2. Note that not all externalities should be proxied, such as `java.io.FileDescriptor` and `java.lang.Runtime` for reasons listed in the table. Our prototype includes implementations for each of the listed externalities other than `RandomAccessFile`, `Runtime` and `Process` because the applications we have tested so far in Flexible JAMM do not use those.

Some native methods are **private**, meaning they can only be invoked by objects of the class in which they are defined. These are indirectly invoked by other objects via a **public** method that in turn calls the private native method. In our prototype, we override this behavior at the level of the public Java method. We modify the public method so that it retrieves data differently depending on whether the externality is shared. If the externality is not being shared, the native method is invoked as before, accessing the resource locally, otherwise a proxy is used to access the resource remotely.

```
 1.   boolean shared = false;

 2.   public boolean inShareMode() {
 3.     return shared;
      }

 4.   private void setShared(boolean value) {
 5.     shared = value;
      }

 6.   FileInputStream proxy = null;

 7.   public int read() throws IOException {
 8.     if (inShareMode())
 9.       return proxy.read();
10.     else
11.       return readNative();
      }

12.   private native int readNative() throws IOException;
```

Figure 13: Code to replace the original public native `FileInputStream.read()` method. If the `FileInputStream` is being used within a shared application, the proxy will be accessed (line 9). Otherwise, the private native method, `readNative`, will be accessed (lines 11 and 12).

A **public native** access method, such as `FileInputStream.read()` cannot be as easily modified at the Java level, because the public native implementation is executed directly when the method is invoked. For these cases, we "privatized" the original public native method, renaming it in the form <*originalName*>`Native`. Then the method with the original name is turned into a non-native, Java method and follows the bridge pattern described in the preceding paragraph. For example, the code seen in Figure 13 replaced the public native `FileInputStream.read()` method.

4.2 Instantiating an Externality

When running a shared application, the system must determine if a newly constructed externality should be accessed locally or remotely. This depends on whether the externality is constructed by the application or by the Java virtual machine which needs to access unshared externalities, such as files (e.g., to obtain class bytecode) and the system time (e.g., to determine when to execute garbage collection). Therefore, the system only creates proxies for an externality that is instantiated by objects in a shared application.

To test for this, Flexible JAMM uses a class loader to scope the application classes in a manner similar to how Java applet security determines whether to allow access to a restricted resource. Flexible JAMM loads all application classes via an implementation of `java.lang.ClassLoader` and when an application is shared, a flag is set in the application's `ClassLoader`. When an application instantiates an externality, Flexible JAMM needs to determine if the application is shared. Externality classes are not loaded in the application class loader, but in the system class loader. Therefore, we use a `SecurityManager` to obtain the object's call stack. Then, if any class loader in the stack is set to share mode, the externality is a descendent of a shared application and therefore constructs a proxy and an externality server.

When constructed, an externality class queries the Flexible JAMM security manager to determine if it should construct a proxy or local externality. The security manager in turn checks the class loader of each object on the execution stack. If any class loader in the stack is set to share mode, then the calling object is a descendent of a shared application and the externality constructs a proxy. If the shared externality is being instantiated by the master replica, then Flexible JAMM's proxy manager creates both a server and a proxy. The proxy manager then sends a message to all replicas containing a reference to the externality server. On each replica, the proxy manager creates a proxy and waits for the reference to the externality server to arrive. Once the reference arrives, the replica's proxy connects to the externality server.

5 Future Work

As a distributed system, our approach contains problems common to such. Although we have considered common problems in the context of the unique aspects of this system there is more to be done. For example, our design addresses recovery from faults on master and non-master proxy connections but it does not address the full range of issues related to fault tolerance. The use of proxied externalities does not, in general, add new problems related to fault tolerance over what are already present with any distributed system and an obvious area of future work involves integrating known solutions and investigating new approaches to fault tolerance with respect to proxied externalities.

Another area of exploration involves relaxing the restriction that replicas must make exactly the same requests in the same order. We can imagine situations where it would be useful to have replicas access shared externalities in a non-uniform manner. Benefits may include improved efficiency for individual replicas (speed, network bandwidth, etc.), or better support of concurrent independent work for users and the relaxation of WYSIWIS. Such a capability would likely require the replicas to specify the data they desire more precisely than using typical read and write requests. The replicas might need to specify the version of the externality, or the state of the replica when making the request. The externality server would return data appropriate to the version, replica state, and possibly other parameters. Such a capability may be highly application-dependent and therefore less generally applicable than what is described here.

6 Summary and Conclusions

Applications commonly acquire input from and write output to data resources external to the application, such as files, databases, sockets, and the system clock. We described common problems associated with sharing externalities in replicated synchronous collaborative applications. We also described a range of *ad hoc* and two general solutions: full literal replication and a novel approach of using replicated proxies to access a centrally located externality. In contrast to the *ad hoc* approaches, the proxy approach allows, and indeed depends on, all replicas to access externalities *uniformly* by making the same requests in the same order. Thus, the proxied-externalities approach is particularly well suited for use in a replicated application-sharing system where each replica is identical. The approach is also applicable in replicated collaboration-aware applications as long as each replica accesses externalities via the same calls in the same order. The replicas may behave differently in every respect other than how they access externalities, thus potentially allowing relaxed WYSIWIS and independent concurrent work.

We described a prototype implementation of this approach within an application-sharing system, called Flexible JAMM. In our prototype, we extended the general design to allow an externality to switch from direct, local access to proxied, remote access when an application is switched from single- to multi-user mode. We treated the system clock specially so that queries by the virtual machine always access the local machine time but queries from shared application objects access the time via a proxy. We described how Flexible JAMM determines whether to create a proxy or access a local externality directly when an object of an externality class is instantiated.

The primary contribution of the work reported here is a systematic solution to the problem of accessing data that are external to a replicated shared application. The use of proxied externalities benefits replicated synchronous groupware in a two key ways. The first is that proxies can be accessed using code similar to that used in traditional, non-distributed, single-user applications. This capability allows a replicated application-sharing system to replace a reference to an actual externality with a proxy transparently to the application. This makes replicated architectures, and the accompanying network resource optimizations and usability benefits, more viable for application sharing. In addition, proxied externalities simplify the development of collaboration-aware applications by allowing the programmer to use the same techniques used in traditional single-user applications. The second key benefit is that development complexity is reduced by programming the same behavior in all replicas. The programmer does not write special externality access code for replicas acting in different roles (master or slave) nor does she need to designate or manage which replicas are acting in which role. The designation and management of roles are handled in the proxies, decreasing complexity at the application level and allowing the developer to address other critical issues in the creation of a collaborative application. This general approach to handling externalities lowers the cost of synchronous groupware development, thereby advancing the trend toward inter–personal computing.

7 Acknowledgements

We thank the reviewers for their thoughtful and constructive suggestions. We gratefully acknowledge the support of Sun Microsystems, Inc. and the National Science Foundation under grant DGE-9553458.

References

Begole, James "Bo", Struble, Craig A., Shaffer, Clifford A., & Smith, Randall B. 1997. Transparent Sharing of Java Applets: A Replicated Approach. *Pages 55–64 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-97).* New York: ACM Press.

Begole, James "Bo", Rosson, Mary Beth, & Shaffer, Clifford A. 1998. Supporting Worker Independence in Collaboration Transparency. *Pages 133–142 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-98).* New York: ACM Press.

Begole, James "Bo", Rosson, Mary Beth, & Shaffer, Clifford A. 1999. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. *ACM Transactions on Computer–Human Interaction (TOCHI)*, **6**(2), 95–132.

Begole, James M.A. 1998 (Dec.). *Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems.* Ph.D. thesis, Virginia Polytechnic Institute and State University, Department of Computer Science.

Burridge, Rich. 2000. *Java Shared Data Toolkit User Guide.* User Guide. Sun Microsystems, Inc., Mountain View, CA. <http://java.sun.com/products/java-media/jsdt/>. last accessed: Apr. 29, 2001.

Chabert, Annie, Grossman, Ed, Jackson, Larry S., Pietrowiz, Stephen R., & Seguin, Chris. 1998. Java Object-sharing in Habanero. *Communications of the ACM*, **41**(6), 69–76.

Coulouris, George, Dollimore, Jean, & Kindberg, Tim. 1994. *Distributed Systems: Concepts and Design.* Second edn. Reading, MA, USA: Addison-Wesley.

Crowley, Terrence, Milazzo, Paul, Baker, Ellie, Forsdick, Harry, & Tomlinson, Raymond. 1990. MMConf: An Infrastructure for Building Shared Multimedia Applications. *Pages 329–342 of: Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work.* Systems Infrastructure for CSCW. Los Angeles, California: ACM Press.

Dewan, Prasun. 1999. Architecutres for Collaborative Applications. *Chap. 7, pages 169–193 of:* Beaudouin-Lafon, Michel (ed), *Computer-Supported Cooperative Work, Trends in Software Series.* John Wiley & Sons.

Dourish, Paul. 1998. Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications. *ACM Transactions on Computer–Human Interaction (TOCHI)*, **5**(2), 109–155.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1996. *Design Patterns: Elements of Reusable Object-oriented Software.* Reading: Addison Wesley.

Graham, T. C. Nicholas, Urnes, Tore, & Nejabi, Roy. 1996. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. *Pages 1–10 of: Proceedings 9th Annual ACM Symposium on User-Interface Software and Technology (UIST-96).* New York: ACM Press.

Greenberg, Saul, & Roseman, Mark. 1999. Groupware Toolkits for Synchronous Work. *Chap. 6, pages 135–168 of:* Beaudouin-Lafon, Michel (ed), *Computer-Supported Cooperative Work, Trends in Software Series*. John Wiley & Sons.

Grudin, Jonathan. 1994. Computer-Supported Cooperative Work: History and Focus. *Computer*, **27**(5), 19–26.

Lantz, K. 1986. An Experiment in Integrated Multimedia Conferencing. *In: Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '86).* Austin, Texas: ACM Press. Reprinted in I. Greif (editor), *Computer-Supported Cooperative Work:A Book of Readings,* pp. 533–552, Morgan Kaufmann, 1988.

Lauwers, J. C. 1990. *Collaboration Transparency in Desktop Teleconferencing Environments*. Ph.D. thesis, Computer Systems Laboratory, Stanford, CA.

Lee, Jang Ho, Prakash, Atul, Jaeger, Trent, & Wu, Gwobaw. 1996. Supporting Multi-User, Multi-Applet Workspaces in CBE. *Pages 344–353 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Mowbray, T. J., & Zahavi, R. 1995. *The Essential CORBA: Systems Integration using Distributed Objects*. Canada: Wiley.

OpenOffice.org. 2001. *OpenOffice.org*. <http://www.openoffice.org/>. last accessed: May 14, 2001.

Patterson, J.F., Day, M., & Kucan, J. 1996 (Nov.). Notification Servers for Synchronous Groupware. *Pages 122–129 of: Proceedings of CSCW '96.*

Patterson, John F. 1991. Comparing the Programming Demands of Single-User and Multi-User Applications. *Pages 87–94 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'91)*. CSCW. New York: ACM Press.

Prakash, Atul, & Shim, Hyong Sop. 1994. DistView: Support for Building Efficient Collaborative Applications using Replicated Active Objects. *Pages 153–164 of: Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*. Collaborative Editing and Reviewing.

Reinhard, Walter, Schweitzer, Jean, Volksen, Gerd, & Weber, Michael. 1994. CSCW Tools: Concepts and Architectures. *Computer*, **27**(5), 28–36.

Roseman, Mark, & Greenberg, Saul. 1996. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer–Human Interaction (TOCHI)*, **3**(1), 66–106.

Schuckmann, Christian, Kirchner, Lutz, Schümmer, Jan, & Haake, Jörg M. 1996. Designing Object-Oriented Synchronous Groupware With COAST. *Pages 30–38 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Sessions, Roger. 1997. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Son.

Stefik, M., Bobrow, D. G., Foster, G., Lanning, S., & Tatar, D. 1987. WYSIWIS Revised: Early Experiences with Multiuser Interfaces. *ACM Transactions on Office Information Systems*, **5**(2), 147–167.

Strom, Banavar, Miller, Prakash, & Ward. 1998. Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects. *IEEE Transactions on Computers*, **47**.

Sun, Chengzheng, & Ellis, Clarence (Skip). 1998 (Nov.). Operational Transformation in Real-Time Group Editors: Issues Algorithms, and Achievements. *In: Proceedings of the ACM conference on Computer-Supported Cooperative Work (CSCW'98).*

Wollrath, Ann, Riggs, Roger, & Waldo, Jim. 1996. A Distributed Object Model for the Java System. *Computing Systems*, **9**(4), 265–290.