Resource Sharing for Replicated Synchronous Groupware

James Begole, Randall B. Smith, Craig A. Struble, and Clifford A. Shaffer, *Member, IEEE*

***Abstract*—We describe problems associated with accessing data resources external to the application, which we term *externalities*, in replicated synchronous collaborative applications. 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 or centralized collaborative systems. We describe *ad hoc* solutions that add to development cost and complexity because the developer must program different behavior for different replicas.**

We introduce a novel general approach to accessing externalities uniformly in a replicated collaborative system. The approach uses a semireplicated architecture where the actual externality resides at a single location and is accessed via replicated *proxies*. This approach allows developers of replicated synchronous groupware to 1) use similar externality access mechanisms as in traditional single-user applications, and 2) 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 and discuss the tradeoffs of this semireplicated 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 Multiuser).

***Index Terms*—Collaborative work, concurrency control, distributed computing, file servers, object-oriented programming, software.**

I. Introduction

WITH TODAY'S proliferation of electronic devices and near universal networking, the emphasis on *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 major contributor to this deficiency is the cost of including support for synchronous collaboration in an application. Several technical and human factors must be addressed that are not necessarily present in a single-user application such as the distribution architecture, concurrency control, and collaborative usability [12], [18]. 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 [3], [4], [8], [10], [21], [15], which facilitate the creation of synchronous multiuser 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. We also describe *ad hoc* and general solutions, introducing a novel semireplicated solution in which the actual externality is accessed via replicated proxies. To our knowledge, this is the first time that the range of problems surrounding externalities in replicated groupware has been addressed explicitly or that this general solution has been presented.

Manuscript received September 18, 1999; revised March 19, 2000 and April 21, 2001; recommended by IEEE/ACM TRANSACTIONS ON NETWORKING Editor B. Plattner. This work was supported by Sun Microsystems, Inc., and by the National Science Foundation under Grant DGE-9553458.

J. Begole and R. B. Smith are with Sun Microsystems Laboratories, Palo Alto, CA 94303 USA (e-mail: Bo.Begole@Sun.com; Randall.Smith@Sun.com).

C. A. Struble is with the Department of Computer Science, Marquette University, Milwaukee, WI 53201 USA (e-mail: craig.struble@marquette.edu).

C. A. Shaffer is with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 USA (e-mail: shaffer@shaffer.cs.vt.edu).

Publisher Item Identifier S 1063-6692(01)10543-1.

II. Groupware Architectures

Synchronous collaborative systems are generally distributed. 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 [13], [5]. The diagrams in Figs. 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.

A. Architecture Tradeoffs

The key advantage of centralized architectures is that they have no problem with data consistency because there is only one copy. On the other hand, they typically require higher network bandwidth to distribute graphical display information than does a replicated implementation which can distribute only minimal update information. 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 [24], 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 process. Finally, centralized approaches are potentially less fault tolerant than replicated, because the central host is a single point of possible systemwide failure.

A *purely* centralized architecture is not possible in practice because, minimally, a representation of the shared data must be replicated to each participating user. Therefore, all synchronous collaborative systems are in fact semi- to fully replicated. The

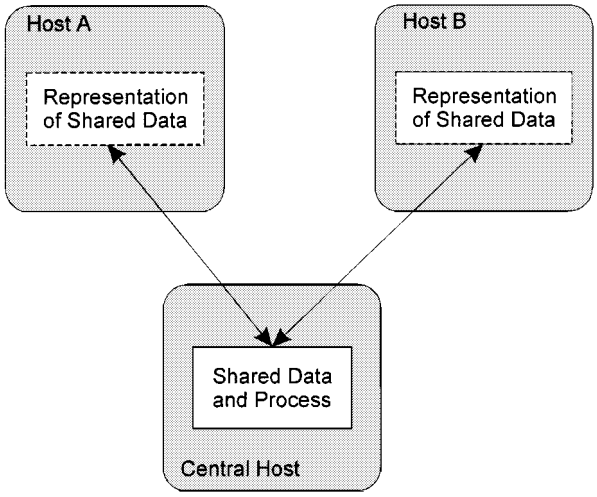

Fig. 1. Centralized architecture. The shared data and process, indicated by the solid-outline box, exist at a single host. A user at a remote host views and manipulates the data via a representation, indicated by a dashed box. Arrows indicate network traffic.

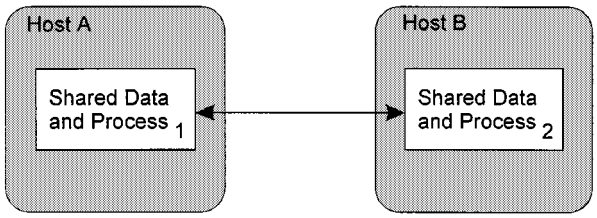

Fig. 2. Replicated architecture. Each host contains a full copy of the shared data and process.

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 [7].

Replicated and semireplicated architectures have the advantage of potentially lower bandwidth and support for collaborative usability principles. First, a replicated system can provide faster response to user input by updating the local copy 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 also supported by allowing participants to modify their local copy of data and merging the changes with remote copies semisynchronously using techniques such as operational transformation [26]. However, maintaining consistency among 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 [2], [11].

One class of real-time groupware that generally does not use replication 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 [1], [19], [20], [22].

B. Shared Resource Problems

Computers have several sources of input and destinations for output: keyboard, mouse, screen, printers, files, databases, network connections, other processes, system time, environment variables, etc. We use the term *externality* for a source of input or output—other than user input and display output—that is external to the application. We exclude user input–output because the problems of handling them are fundamentally different from those of other input–output resources, and in many ways the problems are reversed. Inputs generated by multiple users must be *merged* in some fashion. Conversely, input originating from a single externality is *multicast* to multiple replicas. Display output originating from the shared application is likewise multiply displayed for each user, whereas output to be written to an externality generally should go only to one instance, as we will discuss shortly. Solutions to issues surrounding user input–output can be found in the literature [1], [11], [26], 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, multiple replicas should not access an individual externality directly, either because they do not have access, 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, consider a replicated application which reads a file, say `bookmarks.html`. If a file of the same name resides on each host but contains different data, each replica would receive different input. Different input can lead to replica inconsistency because, as copies of the same deterministic process, we can only guarantee consistency when all replicas receive the same input. Therefore, the replicas must run in *effectively* the same environment. Techniques to provide the illusion that replicas share one environment are described in the next section.

In some cases, however, an application may behave incorrectly if all replicas receive the same input from a particular externality. For example, applications may use unique aspects of the local environment such as the user's home directory, current working directory, and command path. A replica may behave incorrectly if it is given a value from a different replica's environment. For example, if replica A running on one user's machine requests the value of the user's home directory and is given the home directory of a different user running replica B on a different machine, replica A may not be able to access that directory. Developers 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 may be idempotent in that it would be acceptable for each replica to generate output redundantly. In other cases, however, generating the same output multiple times is not desired. For example, although it would be acceptable for each replica to

write a copy of a file on its local host, it would be annoying if each replica 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.

III. Externality Replication

Externalities are trivially handled under centralized architectures but are more difficult under the replicated architectures favored by groupware toolkits and groupware applications. This section describes the tradeoffs of *ad hoc* and general solutions. Current groupware toolkits (surveyed in [2]) provide no abstractions to facilitate replicated input to, or output from, externalities. *Ad hoc* approaches require replicas to access externalities in a nonuniform way, resulting in complex, error-prone code. In contrast, two general approaches alleviate coordination issues by providing uniform access to externalities: 1) full environment replication, and 2) semireplicated proxies.

A. Ad Hoc Solutions

In some cases, it is possible for all replicas to access a single instance of the externality. One example are resources accessed via a Uniform Resource Locator (URL). Replicas of a multiuser whiteboard application, for example, could use this approach to load clip-art image files from the World Wide Web.

Often, though, an externality is only accessible by one replica, such as when a particular file resides on only one host and is not network accessible. Another problem arises if all hosts have access to an externality locally, such as the system time, but the local states differ. A solution to both of these problems is to designate one host as the source of a particular externality. That replica may be referred to as the "master," and the other replicas may be called "slaves." Fig. 3 illustrates this approach.

1) Explicit Distribution: Consider a replicated multiuser text editor that reads an input file and appends the file contents to an in-memory document. Fig. 4 shows a pseudocode 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 [21], and the Java Shared Data Toolkit[1] [3]) or may be implemented *ad hoc*.

2) Implicit Distribution: The above example requires development of a message protocol and explicit message passing. The sender creates a message and explicitly sends it, and each recipient parses the message and handles it appropriately. Each type of message must have a unique identifier to indicate the behavior the recipient should perform. For example, in Fig. 4, the program performs one action [`appendLocal()`] upon receipt of an `appendText` message, and performs a different action for a `anotherMsgType` message.

[1]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.

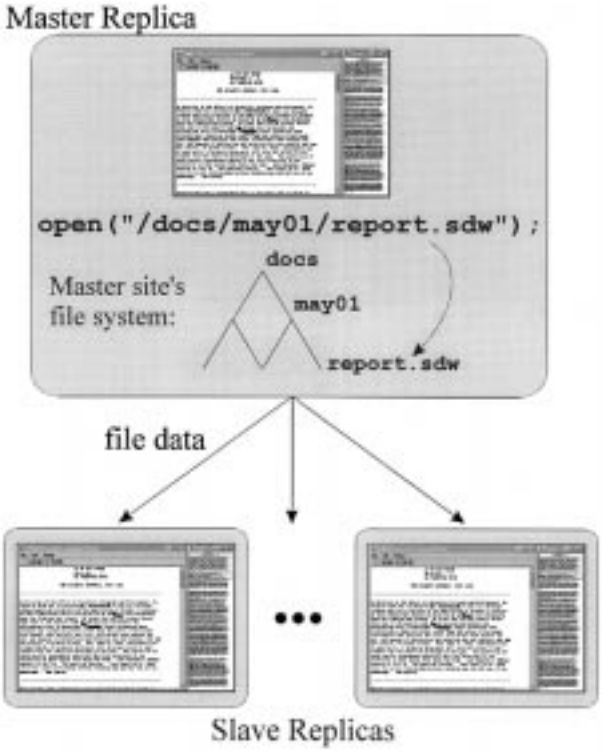

Fig. 3. Illustration of a shared editor application that explicitly distributes externality data among replicas. The "master" replica accesses a file and distributes the contents to the "slave" replicas.

```
 1.  procedure readFile(inFile) {
 2.    if this is the master {
 3.      while (inFile is not empty) {
 4.        read data from inFile and store in buffer
 5.        appendLocal(buffer);
 6.        sendMessageToOthers("appendText", buffer);
     } } }
 7.  procedure receiveMessage(msgType, msgData) {
 8.    if (msgType equals "appendText") {
 9.      read data from msgData and put in inputChars
10.      appendLocal(inputChars);
11.    } else if (msgType equals "anotherMsgType") {
12.       // do something else ....
     } }
13.  procedure appendLocal(inputChars) {
14.    append inputChars to in-memory document
     }
```

Fig. 4. Sample pseudocode to read a file into a replicated collaborative text editor. Data are read by the master replica, appended locally, and then sent to slave replicas (lines 1–6). When the message is received by each slave, `receiveMessage()` is invoked (line 7). 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()` (line 10). Lines 1–6 and 13–14 are invoked on the master, 7–14 on slave replicas.

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) [16], distributed component object model (DCOM) [23] and Java remote method invocation (RMI) [27]. A groupware extension to RPC is provided by GroupKit [21], called multicast remote procedure call (MRPC), which adds the capability to make the invocation on *multiple* remote processes simultaneously. Using MRPC, the

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

Fig. 5. Sample pseudocode using a multicast remote procedure call to directly invoke the procedure that appends data to the document on all replicas. Lines 1–5 are invoked only by the master, 6–7 by all replicas (master and slave).

twelve lines of pseudocode in Fig. 4 may be simplified to the seven lines seen in Fig. 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.

B. General Solutions

The primary disadvantage of the preceding approaches is that the developer must program different behavior for the master and slave replicas. This is prone to error and contributes to the cost of adding real-time collaboration capabilities to an application. There are general solutions, however, that allow the same behavior in all replicas and use similar mechanisms as in single-user nondistributed applications.

Section III-B-1 describes a straightforward approach to handling externalities: full replication. Although full replication can be effective for files, it cannot be applied to all externalities and is therefore not a complete solution. In Section III-B-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.

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 [6], a replicated groupware toolkit, and Dialogo [14], 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 nonfile 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.

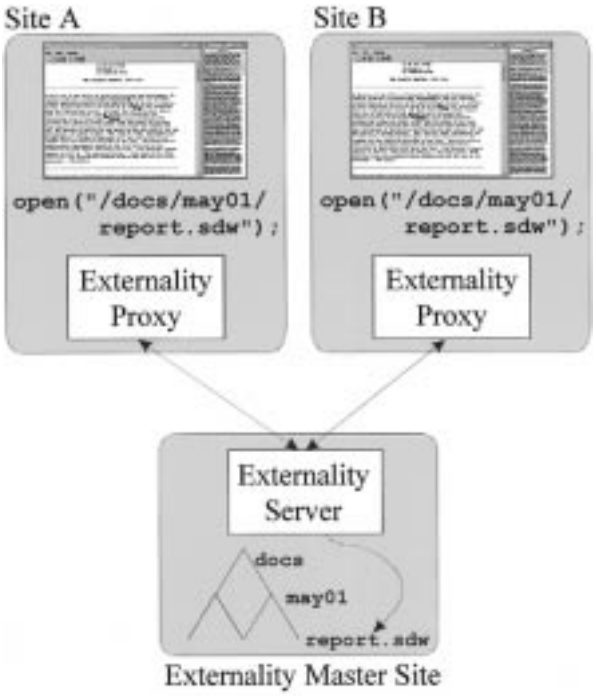

Fig. 6. Shared editor replicas access the externality uniformly via proxies. The externality resides physically on a central site and is accessed by proxies at each replica.

One advantage of literal replication is that it provides an amount of fault tolerance by allowing users to continue working independently if the network fails. In some cases, however, this advantage may be offset by the need to merge conflicting edits later. In any case, from the perspective of the isolated collaborators, the synchronous collaboration is disrupted.

2) Proxied Externalities: This section introduces a semireplicated approach where the externality resides physically at a single location and is accessed via replicated *proxies* [9] that multiplex input to and output from the actual externality. To the application, the proxy acts in place of the actual externality and programmers access the proxy with the same code they would use to access the externality itself.

Fig. 6 shows the architecture for a proxied externality which consists of a client, called the externality proxy, and an externality server. 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 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 requests 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 request parameters to the externality

```
class ReadOnlyExternalityServer {
  LocalReadOnlyExternality realExtern;

  RetType readRequest(Proxy proxy, int requestNumber,
                  Type1 param1, Type2 param2, ...) {
    RetType readResults;
    if proxy has the highest requestNumber {
      readResults = realExtern.readRequest(param1,
                                    param2, ...);
      store request, readResults, and parameters
          in cache keyed by the requestNumber;
    } else {
      access cache table by requestNumber
      readResults = stored return value;
      add proxy to list of proxies that have
          made this request;
    }
    if proxy is last replica to make the request {
      remove requestNumber's data from cache
    }
    return readResults;
} }
```

Fig. 7. Server-side pseudocode for a read request of a proxied input-only 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 so 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 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. Fig. 7 contains pseudocode summarizing how an input-only externality server handles read requests.

After the first proxy makes a particular request, the server may pre-send the result to other proxies in anticipation of their requests, accelerating the response to their request [17], [25]. When a replica leaves normally, the proxy notifies the server which discontinues tracking that replica's requests. If a replica is separated from the session abnormally, 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.

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 its write requests are actually sent to the externality server and written to the externality. The designation of the master may depend on which host has access to the externality or other criteria. All write requests made by other replicas are ignored. However, all proxies store the write requests locally so that each is able to take over as master in case the master is cut off. In such a case, a distributed consensus algorithm designates a new master which 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 pseudocode in Fig. 8 summarizes how an output-only externality proxy handles write requests.

```
class OutputOnlyProxy {
  OutputOnlyExternalityServer outputOnlyServer;

  void writeRequest(Type1 parm1, Type2 parm2, ...) {

    increment requestNumber;

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

Fig. 8. Proxy-side pseudocode for a write request of a proxied output-only externality. Only the master's request is actually sent.

```
class ReadWriteExternalityServer {
  LocalReadWriteExternality realExtern;

  RetType readRequest(ProxyID proxy,int requestNumber,
                  Type1 param1, Type2 param2, ...) {
    RetType readResult;
    if requestNumber is more than 1 greater than last
      requestNumber for this proxy {
      // synchronize proxy with master
      while requestNumber is more than 1 greater than
        the master's last requestNumber {
        block this proxy until master catches up
            and calls readRequest
      }
      retrieve requestNumber's data from cache
      readResult = cached data;
      add proxy to list of proxies that have
          made this request;
    } else if proxy has highest requestNumber {
      readResult = realExtern.readRequest(param1,
                                    param2, ...);
      store request, readResult, and parameters
          in cache keyed by requestNumber;
    } else {
      retrieve requestNumber's data from cache
      readResult = cached data;
      add proxy to list of proxies that have
          made this request;
    }
    return readResult;
} }
```

Fig. 9. Server-side pseudocode for a read request of a proxied input-and-output externality. Nonmaster proxies set the checkSynch value to true only for read requests that follow a write.

Input-and-output externalities, such as read–write files or databases, are handled by combining the above two approaches, summarized by the pseudocode for an input-and-output externality server in Fig. 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. 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.

Synchronization is performed in the following way. Recall that each proxy increments its request number by one with each

TABLE I
A SERIES OF READ AND WRITE REQUESTS SENT FROM TWO PROXIES TO AN INPUT-AND-OUTPUT EXTERNALITY SERVER

		Proxy A (Master)		Proxy B (Slave)	
Time	Value	Req. Num	Operations	Req. Num	Operations
1	5	1	`setValue(5)` (This is sent and executed because A is the master.)		
2	5	2	`x = getValue()` (5 is returned and is cached, associated with req # 2.)		
3	5			1	`setValue(5)` (This is not sent because B is not the master.)
4	5			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.)
6	5			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.)
7	6	3	`setValue(x+1)` (This is sent and executed because A is the master.)		
8	6			4	(Now req # 4 is only 1 greater than the master's last req #. So, the server returns 6 and caches it.)
9	6	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.)		
Final	6	`x==5, y==6`		`x==5, y==6`	

read and write request. When a nonmaster 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 intervening write requests were not sent. When the externality server receives a request, it checks to see if the difference between this request number and that proxy's previous request number is greater than one. If so, the server needs to synchronize this proxy with the master before returning the value of the read, which is done 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 request issued by the proxy so the externality's state is up to date. 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.

As an example, consider an externality with only two operations: `void setValue(int newValue)` sets the value of the externality; and `int getValue()` 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 I traces how the server responds to two proxies issuing this series of operations.

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[14] and Flexible JAMM [1]. Each replica makes the same requests in the same order, because each replica is a copy of the same deterministic process.

Proxied externalities may also be used in 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 are not required to behave uniformly in any other respect. Generally, replicas differ in their views of shared data, not in how they acquire or store the data.

Because this semireplicated system has a centralized component it carries two disadvantages common to centralized architectures. The first is that proxied externalities are less fault tolerant than full literal replication (Section III-B-1). Under the proxied approach, a user cannot continue 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

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

Fig. 10. Sample pseudocode using a proxied input file. The code is similar to that of a traditional single-user application and all replicas use the same behavior.

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 again 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. 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.

4) Benefits: 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 II-B. 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 [1]. Proxied externalities make replicated architectures more viable for application-sharing systems which can alleviate the usability problems found in conventional, centralized systems.

This approach also benefits collaboration-aware applications. Externalities are accessed using code similar to that used in a traditional single-user application. Additionally, at the application level, all replicas have the same behavior because the master–slave coordination is managed in the proxies. These simplifications can result in faster development and more reliable code. As an example, consider that whereas the *ad hoc* approaches described in Section III-A used minimally two methods consisting of the seven lines of pseudocode (not counting master/slave management) seen in Fig. 5, the proxied approach can use one method consisting of the four lines shown in Fig. 10.

How much savings there would be in lines of real code, as opposed to pseudocode, depends on the language and platform 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 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 externality content distribution.

IV. Prototype Implementation

We have implemented a prototype of our proxied externality approach as part of a replicated application-sharing system for the Java platform, called Flexible JAMM (Java Applets Made Multiuser) [1]. To allow transparent, dynamic replacement of an externality with a proxy, we modified core library and native platform classes in Sun's Java 1.1.6 runtime environment. As a result, the prototype uses a nonstandard Java runtime environment.

Fig. 11 shows our class design in the proxy and server implementation for a Java read-only file resource, `java.io.FileInputStream`. The proxy implements the `Proxy` interface which defines the method called `connectToMaster()` which is called to connect to the server and associate this proxy with a unique identifier. The server uses this identifier to keep track of each proxy's requests. The `ProxyFileInputStream` class contains a `remoteResourceLocator` which contains the address of the externality server to which this proxy will connect. The `ProxyFileInputStream` also contains a reference to an interface for the remote externality server, `RemoteFileInputStream`, which implements `RemoteExternality`, which defines a method by which proxies register [`registerProxy()`] and a method that proxies can use to create a unique identifier [`getConnectionNumber()`]. `RemoteFileInputStreamImpl` is an implementation of the actual externality server.

We modified the original class to contain a reference to either an externality proxy or a local externality which may be switched dynamically. This approach follows the *bridge* design pattern, described by Gamma *et al.* [9], which decouples an abstraction (e.g., `FileInputStream`) from its implementation (e.g., a physical file), allowing the implementation to change at run time. This allows the externality to be switched between single- and multiuser access. 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 the local externality to a remote externality server. The original externality is wrapped by the externality server. Fig. 12 shows an example of (a) 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()`], because it is not only accessed by an application executing within the Java virtual machine (VM) but also by the VM itself. Consistent application 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. Instead, for each access of the system clock, we checked to see if the request came from an application-level object. If

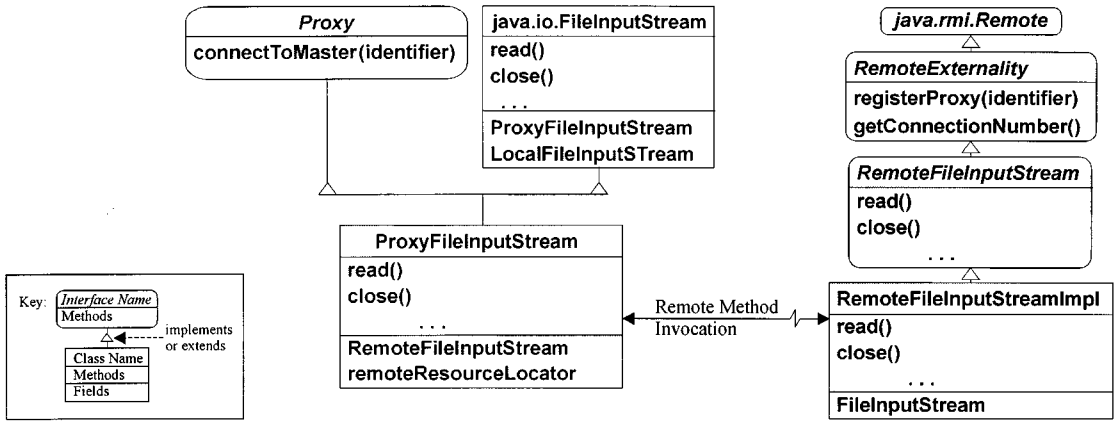

Fig. 11. Class diagram for the proxy and server for the Java input-only file externality, `FileInputStream`.

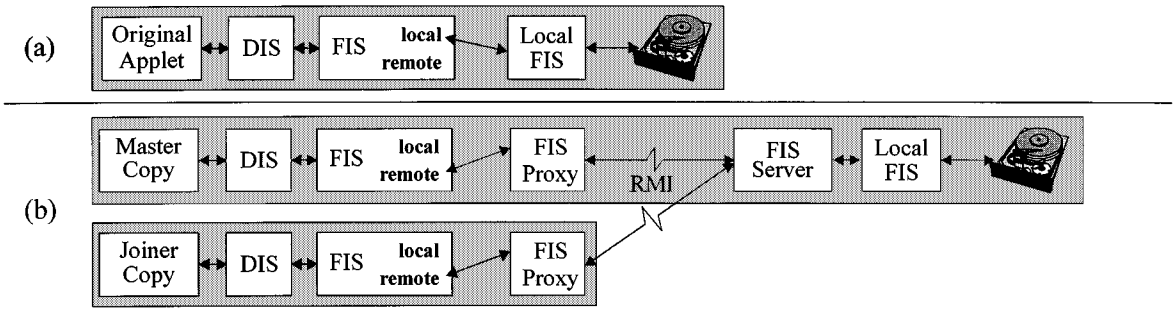

Fig. 12. Switching from single- to multiuser access dynamically. (a) The original applet has a `java.io.DataInputStream` (DIS) connected to a `java.io.FileInputStream` (FIS), which refers to a Local FIS, which reads the physical disk. (b) When the application is shared, the original Local FIS is wrapped by a FIS Server which is accessed via each replica's FIS Proxy.

so, the proxy was accessed, otherwise the local machine was accessed. As a result, sharing `java.lang.System` is more complex than the other externality classes in that it performs an additional check before accessing the data.

A. Java Externality Classes

Identifying externality classes in Java is straightforward, because, as a rule, a Java externality class will access the actual externality via a *native* method, a platform-specific implementation of the method. For example, `FileInputStream` reads from a physical file via a native method named `read()`. Table II lists the externality classes in version 1.1 of the core Java class library. Not all externalities should be proxied as explained in the table. Flexible JAMM includes implementations for each of the listed externalities except `RandomAccessFile` and `Process`, because the applications we tested did not use those.

Many native methods are `private`, meaning they cannot be invoked by applications directly. Typically, applications indirectly invoke these via a `public` method that in turn calls the `private native` method. For these cases, we modified 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.

A `public native` method cannot be as easily modified, because applications invoke the native implementation directly. For these cases, we "privatized" the original `native` method, renaming it in the form ⟨*originalName*⟩`Native`. The original method name becomes a `public` nonnative method, which follows the bridge pattern described in the preceding paragraph. For example, the code seen in Fig. 13 replaces `public native FileInputStream.read()`.

B. Instantiating an Externality

When an externality class is instantiated by a shared application it needs to be accessed remotely. However, when instantiated by the VM (e.g., to obtain class bytecode from a file), the externality needs to be accessed locally. Therefore, the system needs to differentiate these cases. To do so, Flexible JAMM uses an implementation of `java.lang.ClassLoader` similarly to how Java applet security detects whether an access to a restricted resource comes from an applet or the VM. When an application is shared, a flag is set in the application's `ClassLoader`.

When constructed, an externality class queries the Flexible JAMM security manager which 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 the address of the externality server to all replicas. At each replica, the proxy manager

TABLE II
EXTERNALITIES IN VERSION 1.1 OF THE JAVA CLASS LIBRARY

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 nonsequentially.
java.io.File	File contains platform-specific information for path and file separators, and file-level operations other than read and write.
java.io.FileDescriptor	FileDescriptor represents an open file or socket. Applications do not use FileDescriptor objects directly, so this need not 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.

```
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;
```

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

creates a proxy and connects to the externality server once the address of the externality server arrives.

V. FUTURE WORK

As a distributed system, proxied externalities suffer problems common to distributed systems. We have considered such issues in the context of the unique aspects of this system but an obvious area of future work involves integrating known solutions and investigating new approaches to distributed-system problems in 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 nonuniform manner. Benefits may include improved efficiency for individual replicas, or better support of flexible styles of collaboration. Such a capability would likely require the replicas to specify the data they desire more precisely than in the present system. The replicas might need to specify the version of the externality, or the state of the replica when making the request, or possibly other parameters. Such a capability may be highly application dependent and therefore less generally applicable than what we have described here.

VI. SUMMARY AND CONCLUSIONS

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*, making the same requests in the same order. This 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.

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 multiuser 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 externality class is instantiated.

The primary contribution of this work is a systematic solution to the problem of accessing data that are external to a replicated shared application. The use of proxied externalities simplifies development of replicated synchronous groupware in two key ways. The first is that proxies can be accessed using code similar to that used in traditional nondistributed single-user applications. This capability allows a replicated application-sharing system to replace a reference to an actual externality with a proxy transparently. Collaboration-aware application developers also benefit by using the same access mechanisms as 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 code for replicas acting in different roles (master or slave) and does not 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.

This general approach to handling externalities lowers the cost of development for replicated synchronous groupware. The use of replicated architectures brings improved network usage over centralized architectures along with support for relaxed WYSIWIS and independent work. These capabilities allow collaborators to shift naturally between tightly and loosely coupled forms of collaboration.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their thoughtful and constructive suggestions.

REFERENCES

[1] J. Begole, M. B. Rosson, and C. A. Shaffer, "Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems," *ACM Trans. Computer–Human Interaction*, vol. 6, no. 2, pp. 95–132, 1999.

[2] J. M. A. Begole, "Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems," Ph.D. dissertation, Virginia Tech, Dept. of Computer Science, Blacksburg, VA, Dec. 1998.

[3] R. Burridge. (2000) Java shared data toolkit user guide. Sun Microsystems, Inc., Tech. Rep. [Online]. Available: http://java.sun.com/products/java-media/jsdt/.

[4] A. Chabert, E. Grossman, L. S. Jackson, S. R. Pietrowiz, and C. Seguin, "Object-sharing in Habanero," *Commun. ACM*, vol. 41, no. 6, pp. 69–76, 1998.

[5] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*. Reading, MA: Addison-Wesley, 1994.

[6] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, "MMConf: An infrastructure for building shared multimedia applications," in *Proc. ACM Conf. Computer-Supported Cooperative Work*, 1990, pp. 329–342.

[7] P. Dewan, "Architectures for collaborative applications," in *Computer-Supported Cooperative Work, Trends in Software Series*, M. Beaudouin-Lafon, Ed. New York: Wiley, 1999, ch. 7, pp. 169–193.

[8] P. Dourish, "Using metalevel techniques in a flexible toolkit for CSCW applications," *ACM Trans. Computer–Human Interaction*, vol. 5, no. 2, pp. 109–155, 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1996.

[10] T C. N. Graham, T. Urnes, and R. Nejabi, "Efficient distributed implementation of semireplicated synchronous groupware," in *Proc. ACM Symp. User-Interface Software and Technology (UIST'96)*, 1996, pp. 1–10.

[11] S. Greenberg and M. Roseman, "Groupware toolkits for synchronous work," in *Computer-Supported Cooperative Work, Trends in Software Series*, M. Beaudouin-Lafon, Ed. New York: Wiley, 1999, ch. 6, pp. 135–168.

[12] J. Grudin, "Computer-supported cooperative work: History and focus," *Comput.*, vol. 27, no. 5, pp. 19–26, 1994.

[13] K. Lantz, "An experiment in integrated multimedia conferencing," in *Proc. Conf. Computer-Supported Cooperative Work*, 1986, pp. 267–275.

[14] J. C. Lauwers, "Collaboration transparency in desktop teleconferencing environments," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1990.

[15] J. H. Lee, A. Prakash, T. Jaeger, and G. Wu, "Supporting multiuser, multiapplet workspaces in CBE," in *Proc. ACM Conf. Computer Supported Work*, 1996, pp. 344–353.

[16] T. J. Mowbray and R. Zahavi, *The Essential CORBA: Systems Integration Using Distributed Objects*. Toronto, Canada: Wiley, 1995.

[17] J. F. Patterson, M. Day, and J. Kucan, "Notification servers for synchronous groupware," in *Proc. ACM Conf. Computer-Supported Cooperative Work*, Nov. 1996, pp. 122–129.

[18] J. F. Patterson, "Comparing the programming demands of single-user and multiuser applications," in *Proc. ACM Symp. User Interface Software and Technology (UIST'91)*, 1991, pp. 87–94.

[19] A. Prakash and H. S. Shim, "DistView: Support for building efficient collaborative applications using replicated active objects," in *Proc. ACM Conf. Computer-Supported Cooperative Work*, 1994, pp. 153–164.

[20] W. Reinhard, J. Schweitzer, G. Volksen, and M. Weber, "CSCW Tools: Concepts and architectures.," *Comput.*, vol. 27, no. 5, pp. 28–36.

[21] M. Roseman and S. Greenberg, "Building real time groupware with GroupKit, a groupware toolkit," *ACM Trans. Computer–Human Interaction*, vol. 3, no. 1, pp. 66–106, 1996.

[22] C. Schuckmann, L. Kirchner, J. Schümmer, and J. M. Haake, "Designing object-oriented synchronous groupware with COAST," in *Proc. ACM Conf. Computer Supported Work*, 1996, pp. 30–38.

[23] R. Sessions, *COM DCOM: Microsoft's Vision for Distributed Objects*. New York: Wiley, 1997.

[24] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar, "WYSIWIS revised: Early experiences with multiuser interfaces," *ACM Trans. Office Inform. Syst.*, vol. 5, no. 2, pp. 147–167, 1987.

[25] R. Strom, G. Banavar, K. Miller, A. Prakash, and M. Ward, "Concurrency control and view notification algorithms for collaborative replicated objects," *IEEE Trans. Comput.*, vol. 47, pp. 458–471, Apr. 1998.

[26] C. Sun and C. (Skip) Ellis, "Operational transformation in real-time group editors: Issues algorithms, and achievements," in *Proc. ACM Conf. Computer-Supported Cooperative Work*, Nov. 1998, pp. 59–68.

[27] A. Wollrath, R. Riggs, and J. Waldo, "A distributed object model for the Java system," *Comput. Syst.*, vol. 9, no. 4, pp. 265–290, 1996.

James Begole received the B.S. degree in mathematics from Virginia Commonwealth University, Richmond, and the M.S. and Ph.D. degrees in computer science from Virginia Polytechnic Institute and State University, Blacksburg.

He is currently a Research Staff Member in the Network Communities Group at Sun Microsystems Laboratories, Palo Alto, CA. His research involves user interface technologies and design, collaborative computing, computer-mediated communication, and distributed-system architectures.

Randall B. Smith received the Ph.D. degree in theoretical physics from the University of California at San Diego.

He is currently a Principle Investigator at Sun Microsystems Laboratories, where he leads a project on network and data visualization techniques. His interests include distributed computing, user interface issues for virtual shared spaces, object-oriented language design, and interactive distance learning environments.

Craig A. Struble received the B.S., M.S., and Ph.D. degrees in computer science from Virginia Polytechnic Institute and State University, Blacksburg.

He is currently an Assistant Professor of Computer Science at Marquette University, Milwaukee, WI. His current research interests include bioinformatics, computational algebra, data mining, and image processing.

Clifford A. Shaffer (M'82) received the Ph.D. degree from the University of Maryland, Baltimore.

He is currently an Associate Professor of Computer Science at Virginia Polytechnic Institute and State University, Blacksburg. His current research interests are related to developing Problem Solving Environments for engineering and science applications. Specific topics include data structures and algorithms for visualization, collaborative computing, component programming, and user interfaces for specifying models and computations.