Programmer-Controlled Application-Level Multicast

Prasun Dewan
Department of Computer Science
University of North Carolina
Chapel Hill, NC USA
dewan@cs.unc.edu

***Abstract*—Group communication abstractions provide application-level multicasting to communicate information among distributed processes. A variety of such abstractions have been provided to implement synchronous collaborative applications but they do not allow control over the multicast of information to the selected group of processes. We have developed a new abstraction that overcomes this limitation. It defines a two-level grouping of distributed processes, with one level defining the users interacting with a specific collaborative application, and the other defining the set of collaborative applications a set of users is sharing simultaneously to perform some collaborative task. It allows information to be sent directly to the receiving processes or through a centralized relayer. In either case, programmer-choosable and replaceable send and receive filters provide consistency guarantees. The abstraction provides message passing rather than remote procedure calls, and supports asynchronous sending and receiving of messages. It is designed to support both centralized and replicated architectures. The abstraction has been implemented on top of the Java Remote Method Invocation layer and has been used to implement a broad range of collaboration functions.**

Keywords-group communication; collaboration toolkits; multicast; collaboration awareness; consistency; sessions

I. INTRODUCTION

Distributed applications are tedious and difficult to implement as programmers must learn and use either (a) low-level connection details such as establishment, reading and writing of stream abstractions, or (b) complex concepts such as proxies, remote interfaces, remote exceptions, and thread semantics of remote method invocation. In either case, they must be aware of the end points of the parties with which they communicate, and send information to each of them individually. Domain-specific abstractions can ameliorate this situation. In this paper, we focus on the domain of distributed synchronous collaboration.

A variety of abstractions have been developed in the past for this domain. As with other kinds of abstractions, they must balance automation with flexibility – in general, the more tasks abstractions perform for programmers, the less flexibility they offer. Abstractions in this domain have focused on both goals, with abstractions supporting collaboration-awareness and parameterization [1] designed for automation, and those offering group communication abstractions designed for flexibility. We focus on group communication abstractions.

Our specific reason for addressing group communication is to build an experimental research and teaching tested for understanding and improving on the state-of-the art in collaboration concepts. There has been some early pioneering work in such abstractions but it has not evolved significantly for about two decades, and more important, from our point of view, not designed to meet our goal. Previous abstractions support direct communication among processes without enabling any consistency guarantees such as causality, jitter management, or replica consistency. We have developed a new group abstraction to address these limitations. This paper, describes its design, illustrates its uses, and discusses its implementation and our experience with the implementation.

Section 2 discusses the related work on which our abstraction is based. Section 3 presents the design or API of the abstraction, and shows how it can be used in a wide variety of contexts. Section 4 overviews its implementation and use. Section 5 presents conclusions and directions for future research.

II. RELATED WORK

A distributed collaborative application must implement a whole range of functions [2] including: session management, coupling, awareness, access control, and concurrency control. Implementation of each of these functions involves communication among distributed processes, which, as mentioned in the introduction, is non-trivial. Therefore, three forms of abstractions have been offered to ameliorate this problem, which fall at different places in the automation-flexibility spectrum.

Collaboration transparency: These abstractions automatically convert collaboration-transparent single-user programs to collaborative versions [1].

Parameterized: These define a parameterized design space for one or more collaboration functions, and allow application programs to control sharing policies by specifying values for these parameters [1].

Group communication: These allow processes to (a) join and leave collaborative sessions, and (b) "multicast" messages to groups of processes in the session without worrying about or even knowing about the existence of individual members in these groups. This is application-level rather than network-level multicast, as in the underlying network, a separate message is sent to each destination.

Recall that our goal was to create a test-bed that can be used to implement novel collaboration functions and provide students with an implementation-oriented understanding of

existing and novel collaboration functions. Thus, of these three kinds of abstractions, the last seem to be the most appropriate; so let us focus in some depth on them.

The first such abstraction was implemented in the mid-eighties as part of the influential Xerox Colab collaboration environment [15]. For each user in a collaborative session, it created a separate replica of a program. The program was implemented as an extension of an interpretive object-oriented programming language that allowed certain methods to be declared as broadcast methods. Invoking a broadcast method on a replica had the side effect of invoking the same method on all other replicas of the application. The actual task of sending messages to remote processes was handled by this group abstraction, making the application program more or less collaboration- unaware.

Two successor systems, GroupKit [3] and Suite [4], show that it is useful to make the abstraction more flexible. These were developed contemporaneously and independently in the early to mid-nineties, before the advent of Java, and were based on TCL and C, respectively.

GroupKit, like Colab, was designed for the replicated architecture, and supported group invocation of procedures. However, it offered more flexibility in three important ways. First, it allowed a replica to know when a replica of some other user joined or left the collaborative session, supplying the identity of the user, which could be used to provide customized session awareness to users. Second, it allowed a group call to be made not only on all replicas, but also all replicas except the one making the call. Third, the decision about the set of replicas on which a call was invoked was made at runtime by the caller rather than at program writing time by the callee.

Suite was designed for an architecture in which the model or semantics code was centralized and the user-interface code, called dialogue manager, was system-provided and replicated. This architecture allowed Suite to offer collaboration transparency, parameterized collaboration functions, and group communication in a single system. Here we focus on group communication.

The model could make a call in all remote dialogue manager connected to it. It could also make the call in a programmer-chosen group of dialogue managers. When a dialogue manager joined a session, the model was informed of its identity, which could be used to define arbitrary groups of dialogue managers. Finally, a call made by the model in a callback invoked by a dialogue manager could be invoked on two additional predefined groups: all dialogue managers except the one that made the callback, and the dialogue manager that made the callback. The groups defined by Suite are difficult to compare to those defined in GroupKit and Colab as they are designed for a centralized architecture. As we shall see later, it is possible to create a single system supporting both replicated and centralized architectures in which the groups defined by all three systems are included.

III. REQUIREMENTS

Together, the three multicasting primitives surveyed above define a design space of group abstractions in which the four main dimensions consist of the architecture, the groups of processes in which a call is made, and whether the group is decided at program writing time or at runtime, and the awareness a process has about other processes in the session. Based on these choices, GroupKit is more flexible than Colab as it supports caller control over multicasting, an additional multicast group, and awareness of users in the session. In comparison to Suite, it does not impose a centralized architecture. On the other hand, it imposes a replicated architecture as it assumes each communicating process implements the same set of methods so that a method can be called in all of these sites. Thus, none of these systems supports both the centralized and replicated architectures. Both kinds of architecture are useful, as explained in [5]. There has been work in supporting multiple architectures [5] in one system and even adapting the architecture automatically [6], but this work has been done in the context of collaboration-transparency, which as mentioned above, gives programmers no flexibility.

Another important flexibility limitation of existing group communication systems is that they allow no control over the multicast of a message to a group. In particular, they do not allow programmers to (a) determine if a message is sent to a remote process directly or through one or more intermediary processes, and (b) re-order or change messages at the sending and or receiving sites to provide consistency guarantees.

The first property may not seem like a practical limitation. In synchronous collaboration, one can expect direct communication to offer better (remote) response times as the number of processes through which a message passes is minimized. For this reason, to the best of our knowledge, all three systems offer direct communication. However, there are at least three reasons for putting intermediaries.

Response times: Recently, Junuzovic and Dewan [7] have shown that in certain situations, multicasting a message through intermediaries can, in fact, improve remote response times, especially in today's world of wireless communication and mobile computing. To illustrate, imagine a user on a mobile computer on a congested wireless connection making a presentation to a large number of users. In this scenario, response times will be smaller if the mobile computer sends a single message to a more powerful computer on a faster network and the latter relays it to all of the users viewing the presentation.

Firewalls: Often user processes are behind firewalls, which prevent them from communicating with each other directly.

Lock-less Consistency: Certain lock-less consistency algorithms such as atomic broadcast and operation transformation algorithms – in particular the Jupiter operation-transformation algorithm [19] used in GoogleDocs - require messages to be relayed through a server.

This does not mean that communication should always be relayed. When none of these conditions apply, communication should indeed be direct to support faster response times. Thus both forms of communications should be supported. Junuzovic and Dewan [6] have shown it is possible for the system to automatically choose the routing of messages based on

This research was sponsored in part by NSF

response times. However, programmer-control is still necessary to determine if application-specific consistency requirements require relayed communication.

Application-specific consistency requirements also motivate programmer-control over reordering and modification of messages. In direct communication, messages may need to be reordered to ensure causality [8]. In both direct and relayed communication, messages may need to be changed to support operation transformation [9].

Though the importance of these consistency requirements has been known since the first paper on operation transformation in the mid-eighties [10], designers of group toolkits have ignored them. We conjecture that the reason is that these requirements have been motivated mainly for collaborative text editors. To the best of our knowledge, no general-purpose toolkit has targeted such editors, and no other application has implemented (lock-less) consistency Collaborative editors have been implemented mostly by extending existing editors such as Emacs [11] or Word [9] in an editor-specific way without trying to use general purpose language-based abstractions. One exception is the Google Docs editing tool, whose origin is the Writely edtor, which was apparently built ground-up to support collaboration. However, to the best of our knowledge, this was a standalone project and thus did not use or create general-purpose collaboration abstractions.

Given this history and analysis, is it worth addressing (lock-less) consistency in general-purpose abstractions? The answer, we believe is, yes, for four reasons. First, an abstraction cannot be called general purpose if it precludes even a single important class of applications. Second, text editors are provided as part of a whole suite of collaborative applications, and it is important for these applications to reuse as much code as possible. For example, it is important for Google Talk and Google Docs to share code for multicasting messages so that changes to optimize this code are made once for each application. Third, from a teaching or research perspective, it is not so important to have the practical goal of extending existing single-user text-editors - creating such editors ground-up using a collaboration toolkit is a viable alternative. Finally, a text editor is not the only popular application requiring consistency. Arguably, IM, which is part of most collaborative sessions, could also benefit from causality and/or operation transformation, because misinterpretation of concurrent messages as serial can cause problems even in two-person IM. Enabling support for consistency in a general-purpose abstraction will allow a greater variety of applications to offer it. Consistency management is still an active area of research [9], especially as operation transformation algorithms do not come with proofs in which the community believes. Thus, it is important to allow these algorithms to be transparently substituted with possibly programmer-defined ones.

Support for programmer-defined consistency algorithms implies also that there should be a way to test these algorithms, which in turn implies a way to transparently delay delivery to different sites with which a process communicates. Current group abstractions do not provide such control.

As mentioned above, all group communication abstractions must allow processes to join and leave sessions so that multicast groups can be defined based on session membership. However, current systems consider all sessions to be equally related, that is, define a flat hierarchy of sessions. As a result, they do not directly capture modern collaborative environments in which multiple applications such as an IM, text-editing and whiteboard tools are used together by a group of users in a single logical collaborative session. Such environments require a more complex, multi-level session membership and notification semantics.

These, then, are the reasons motivating our project. Our goal is to offer the automation of previous multicast primitive while increasing their flexibility. Table 1 evaluates the current systems against automation and flexibility requirements identified above, and shows that none of the existing abstractions meet all of these requirements. It is our goal to develop an abstraction that meets all of them.

Table 1 System vs. Multicast Requirements

	Colab	GroupKit	Suite	GroupMessages
Direct and Relayed communication	No	No	No	Yes
Caller Control of Message Destination	No	Yes	Yes	Yes
Centralized Architecture	No	No	Yes	Yes
Replicated Architecture	Yes	Yes	No	Yes
Transparent Message Delay, Reordering and Change	No	No	No	Yes
Multi-Application Sessions	No	No	No	Yes

IV. DESIGN AND RUNNING-EXAMPLE

We have designed and implemented a Java-based system, called GroupMessages, to meet this goal. In this section, we describe its design and applications using a running example. We develop the example, incrementally, starting with a single-user program.

A. Model-Based Single-User Program

The single-user program provides a console-based user-interface to echo input lines. It also provides command to view the history of entered input lines and to quit the interactive session, as shown in Figure 1.

This version does not use any of the multicasting primitives. However, for it to be extended to support multi-user interaction, it has to be decomposed in a fashion that allows its

behavior to be extended. Ideally, to offer extensibility, a program should be implemented using appropriate design patterns. One design pattern that applies to all interactive programs is model-view-controller [12], which separates semantics, input, and output of interactive applications into model, view, and controller objects, respectively. Sometimes input and output are so coupled that it is sufficient to implement a coarser-grained version of this pattern in which the view and controller are combined into a single object, which we call an interactor

```
Please enter an input line or quit or history
The woods are lovely,dark and deep
The woods are lovely,dark and deep(Echo)
Please enter an input line or quit or history
```

Figure 1 Single-User Echoer

This is the pattern used in this application (Figure 2). A History model object maintains a list of input lines, and allows other objects to add elements in the list and read the entire list. In response to the add operation, it notifies its observers of this event by calling the elementAdded() operation in them, which in this example, consists of an EchoerInteractor object. This object reads input lines, asks the model to add them, and reacts to a notification from the model by echoing each added line on the screen As we see below, this architecture will allow us to reuse the model and interactor types without adding any collaboration awareness to them, thereby providing vindication for keeping the semantics and user-interface in separate modules. The user-interface of the application is kept simple so that we can focus on the collaborative aspects of the extended application.

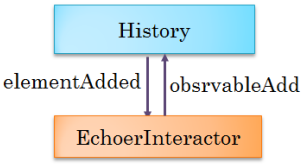

Figure 2 Single-User Architecture

B. Basic Collaborative User-Interface

The collaborative extension of this application is essentially a console-based group IM application, as shown in Figure 3.

```
Please enter an input line or quit or history
The woods are lovely dark and deep
The woods are lovely dark and deep(Echo)
Please enter an input line or quit or history
But I have promises to keep(Bob)
And miles to go before I sleep(Cathy)
```

Figure 3 Echoer to Group IM

Here we see three users, Alice, Bob, and Cathy, using the application. The IM application is a strict extension of the single-user echoer that allows users to see input lines not only entered by them but also their collaborators, and adds to the history both local and remote input. Each user is aware of the identity of the user who entered an input line. A consequence of the awareness is that a user who inputs a line sees a different view of it than the others. This feature has been added to illustrate some of the complications that arise when using multicasting primitives in replicated and centralized architectures.

Thus, we see that this version of our example offers three of the collaboration functions mentioned earlier, session management, coupling and awareness. These functions, of course, are implemented using GroupMessages. Let us first consider session management.

C. Two-Level Session Management

Like GroupKit and Suite, GroupMessages allows processes to ecplicitly create, join, and leave sessions, and be notified when these operations are invoked by processes of other users. As in previous systems, a central process is used to support sessions, which we call the *session server*. Application code interacts with the session server and local multicasting code through a local object called the *communicator*. Figure 4 shows the basic architecture visible to the programmers. They know that a session server exists as they must provide its location. However, all functions of our abstraction are provided through the communicator. The communicator itself is partitioned into several internal components, which the message producers and consumers in the application can ignore. However, as we see later, two of these components are visible to consistency management modules in the application.

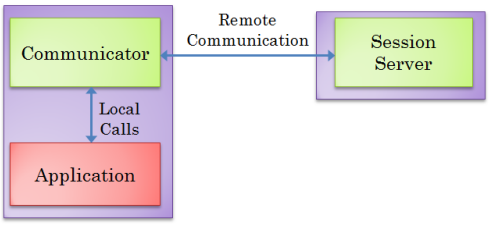

Figure 4 Architecture Visible to the Programmer

Our abstraction accommodates multi-application sessions. A session is not simply a set of users. Instead, it consists of a set of application sessions and users, and each application session consists of a set of users (Figure 5).

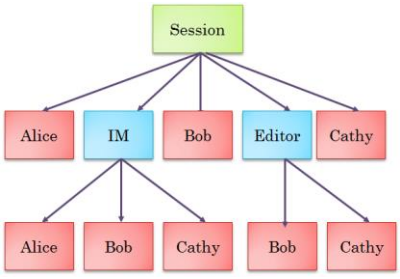

Figure 5 Multi-Level Session Structure

Thus, like other systems, our abstraction allows users to be (dynamically) added to sessions. In addition, it allows applications to be added to sessions, and users to be added to applications. Adding an application to a session creates an *application session*, which corresponds to a session in other systems. Adding users to an application session allows them to use the application to collaborate with other users in the application session. Adding users to a session allows them to be informed of other users and applications in the (overall) session without participating in any joint activity. They can

react to this information by joining one or more application sessions in the (overall) session. In Figure 5, Alice, Bob and Cathy are all in the IM application session, while only Bob and Cathy are in the Editor application session. All three users are in the overall session, and thus, have the option of joining any application session in it. Currently, it is not possible for users to join an application session without also joining the overall session.

A single static call is provided for creating communicators and creating and joining sessions and applications:

```
static Communicator getCommunicator(
        String aServerHost, String aSessionName,
        String aClientName, String anApplicationName,
        String aRoutingKind)
```

Here aServerHost is the name of the host on which the central session server resides, aSessionName is the name of the (overall) session, aClientName is the name of the client making this call, anApplicationName is the name of an application (session), and aRoutingKind denotes whether multicasts through the communicator will be routed through a relayer at the session server. If anApplicationName is null, then the client will be added to the overall session. Otherwise, it will be added to both the application session and the overall session. Of course, if it is already part of a session or application, then it is not added again. If the named session or application does not exist, then it will be created. By combining the creation of sessions and applications with addition of members to them, we allow all replicas in a replicated application to execute the same code and be started in any order. Otherwise, one of them must have special code to create a session and this code must be started before others. The client name is an identifier that distinguishes the caller from other members of a session and application. It can be any string chosen by the programmer. As we see below, our abstraction supports communication with specific clients. This name is used in such communication.

For each application and session combination, a separate communicator is created. Our design expects each user process to be associated with either the overall session or a single application in that session. Thus, we expect each process to create a single communicator, though the design and implementation support multiple communicators if a single process wishes to play the role of multiple logical applications.

Like GroupKit, our system allows processes to receive notifications about successful session creation and joins, including the ones they initiated, by implementing listener methods with the following signature

```
public void clientJoined (String aClientName,
    String anApplicationName, String aSessionName,
        boolean isNewSession, boolean isNewApplication,
        Collection<String> allClients);
```

It is the dual of the getCommunicator() call described above, providing the listener with information about a successful join. The two Boolean flags indicate if the application and session are newly created. allClients is the collection of all previous clients in the session. When a client joins the session, this method is invoked for each existing client and application combination. It is also executed once for each subsequent join. To allow clients to register listeners before they join sessions, a communicator does not automatically join the specified session when it is created. It does so when the client makes a special non-blocking join() call. This call uses the parameters provided at communicator creation time to send an appropriate message to the session server. A communicator also provides a call to leave application/overall sessions, and a notification method to receive information about leaves. Let us use the running example to illustrate the nature of these session functions.

Client Alice creates the following communicator to initiate an application-less joining of session SESSION_NAME:

```
communicator = getCommunicator(SERVER_HOST,
      SESSION_NAME,      ALICE, null,
      Communicator.RELAYED);
```

It then registers a session listener that implements the following method for join notifications:

```
public void clientJoined (String aClientName,
        String anApplicationName, String aSessionName,
        boolean aNewSession, boolean aNewApplication,
        Collection<String> previousClients) {
  if (aNewApplication && anApplicationName != null &&
        IM.equals(anApplicationName))
    joinSession(anApplicationName, aSessionName);
}
```

Here, joinSession() is an internal application method that forks a new process that joins the IM session. Finally, it invokes the join() call. Assuming Alice is the first member of the session, at this point Alice's clientJoined() method will be invoked informing it of the successful execution of the non-blocking join() call. This method does nothing as the application name is null.

Later, client Bob creates a communicator for the IM application:

```
getCommunicator(SERVER_HOST, SESSION_NAME,
    BOB, IM, Communicator.RELAYED);
```

Next it registers a session listener that defines the following join notification method:

```
public void clientJoined (String aClientName,
        String anApplicationName, String aSessionName,
        boolean aNewSession, boolean aNewApplication,
        Collection<String> previousClients) {
    displayMessage(aClientName, anApplicationName);
}
```

When the join call is successfully invoked on the sessiom manager, Alice's clientJoined() method is invoked for Bob; and Bob's clientJoined() method is invoked twice, first for the existing member, Alice, and then for the new member, Bob. Bob's method simply prints a message, while Alice's method forks the process that joins the application session.

If the actions were reversed and Bob joined the IM session before Alice joined the application-less session, the behavior would be more or less the same. Alice's clientJoined() method would still be called for the existing application session. The

only difference is that displayMessage() method in Bob would be called first for Bob and then for Alice. Thus, the semantics are resilient to race conditions arising from uncoordinated join() calls being made by different clients.

Session management functions provides the basis for defining the groups used in multicasting calls. Let us consider these calls next.

D. Message-Based Multicasting

Concurrent systems are often classified as message-based or procedure-based depending on whether they communicate information by sending messages or invoking procedures. As Lauer and Needham [13] point out, these systems are equivalent in expressibility though one might be easier to program in certain situations.

All three group communication abstractions surveyed here are procedure-based in that they multicast (remote) procedure calls. In contrast, GroupMessages, as the name indicates, is message-based, because high-level, efficient and consistent multicasting would have required us to implement our own remote procedure call for Java. Directly using the standard library for Java, RMI, creates several problems:

Transparent syntax: In RMI, a remote method declaration and call has the same syntax as a local method declaration and call, respectively, though the caller has to address new kinds of exceptions. The callee is completely unaware of whether it was invoked remotely or locally, and thus does not know the identity of the caller. To implement awareness, access control, and concurrency control, it is useful to have this information. For instance, in any IM application, a user is aware of the identity of the person who sent a message. To support such awareness, the callers must explicitly send their identities using procedure parameters, even though the underlying system has this information.

Synchronous call: Consistent with its goal of compatibility with local calls, RMI supports synchronous calls, which blocks the caller until the call completes. Experience has shown that these semantics visibly slow down response times when the input rate is fast– in particular when a tele-pointer is moved [14]. The reason is that a sending site must wait for input to reach the remote site and an acknowledgement to return before it can send another input. The problem is aggravated by increasing the message hops; in particular sending the message through a relayer. To achieve concurrency, it is possible to create multiple threads that make synchronous calls – a standard technique in procedure-based systems [13]. However, this adds to the programmer-effort. Moreover, creating multiple threads can tax or exhaust system resources. In particular, in the telepointer case, it is unreasonable to create a thread per move. Thus, the number of outstanding calls would be limited by the size of thread pool used to send the data. Finally, concurrent invocations of a remote method by different threads can lead to consistency problems. To illustrate, assume that in our running example, a user enters two input strings. If these are sent by two different threads, then because of scheduling uncertainties, the second string may reach the destination before the first one. Perhaps for this reason, some implementations do not allow a method to be invoked concurrently by threads in the same site, which leads to the high latency problem mentioned above.

Concurrent remote calls by different sites: In RMI, remote invocations of the same method by different sites execute in different server threads. These calls may need be serialized for consistency reasons. This means that the programmer must be careful to use Java'a (high-level) synchronization mechanisms to provide such serialization

Deadlocks: Synchronous remote invocation and synchronized concurrent remote calls block threads at the invoking and invoked sites, respectively, which in turn can lead to deadlocks. For example, if a synchronized history object in a slave site makes a remote call to a synchronized serialized history object in a master site, and the latter invokes a method back in the slave history to provide a serialized update to the history, then we have a deadlock. This means that programmers must take special steps to avoid such deadlocks.

Single-site proxy: RMI proxies are created at the callee sites and distributed from there to calling sites. They are bound to the creating site. To support multicast RPC, we would have had to change RMI to create proxies at the caller site that forwards calls to multiple programmer--controlled server sites.

Semantics of group function calls: In RMI, a remote method can return a value. Supporting remote function calls requires us to determine what value should be returned by a multicast function call.

These problems are not unique to RMI and also arise if we were to, for instance, use the RPC layer of .NET. These are typical of RPC support for compiled object-oriented languages.

None of the previous multicasting systems change the syntax of call invocation to provide caller awareness. GroupKit does not face the other issues as it is built on top of TCL, which is a scripting interpretive system in which remote void asynchronous procedure calls are made by sending textual representations of the calls, which are simply forwarded asynchronously by GroupKit libraries. Colab is also built on top of an interpretive language (Object Oriented Lisp), and requires (pre)compiler support for labeling methods. The paper on Colab [15] does not address the issues above – in particular it does not indicate if the remote calls are synchronous, and what happens to results of broadcast functions. Presumably, as the functions are guaranteed to execute locally, the local results are returned. The issue of multicast proxies can be handled by appropriate (pre) compiler support. Suite provides multicasting of only predefined void procedures provided by a dialogue manager, which are handled by calling asynchronous remote procedures provide by the Suite RPC layer [16].

GroupMessages uses non-blocking message-based multicasting to address these issues. All outgoing and incoming messages go through the system, which can then control synchrony and threading issues. A client is guaranteed that all outgoing messages to an application session are serialized, as well as all of its incoming notifications. Moreover, a relayer guarantees that for a particular application session, messages leave in the same order in which they arrive.

As RMI is built on top of sockets, a message-based abstraction, it arguably provides a higher-level abstraction than sockets. However, GroupMessages does not have most of the disadvantages of sockets, as programmers do not have to create, bind, and connect sockets or implement threads to read and write from them. It does, however, have the fundamental disadvantage of message-based communication – a program that wishes to make a logical procedure call on a remote site must convert or marshal these parameters into a message at the caller, and unmarshal the message back into parameters at the callee site [17].

Like recent Google APIs, GroupMessages does not offer an explicit call to receive a message. It uses the observer pattern to not only inform interested parties about session notifications but also received data. As a result, programming of synchronous communication is more difficult and must be done using non message passing abstractions such as semaphores and monitors. Our decision is a consequence of the fact that GroupMessages is designed for synchronous collaborative applications in which response times are degraded by blocking.

E. Multcast Groups

The most fundamental multicasting call provided by GroupMessasges is toOthers(), which allows a client that has joined an application session to send an arbitrary data object to all other members of the session. To illustrate, let us continue with our example by outlining how our echoing code was converted into a replicated implementation of the IM user-interface. Figure 6 shows the architecture of this application

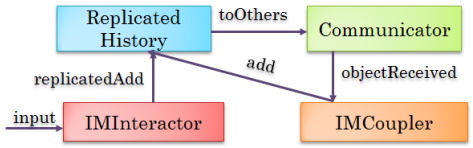

Figure 6 Replicated IM Architecture

The echo model and interactor objects of Figure 2 are replaced by extensions for the replicated IM. When the user inputs a line, the IMInteactor calls replicatedAdd() in the model object. This method calls the observableAdd() method of its superclass, marshals the name and parameters of the add operation into a ListEdit message object, and uses toOthers() to multicast this message to other replicas in the application session:

```
public void replicatedAdd(ElementType anInput) {
        int anIndex = size();
        super.observableAdd(anIndex, anInput);
        ListEdit listEdit = new   AListEdit(
                OperationName.ADD, anIndex, anInput);
        communicator.toOthers(listEdit);
}
```

This message is delivered to a remote replica by calling the objectReceived() listener method:

```
public void objectReceived(Object msg, String clientName) {
   if (msg instanceof ListEdit)
      processListEdit((ListEdit<String>) msg, clientName);
}
```

As we see in the code above, this method has the name of the calling client even though it was not explicitly provided by the caller. This method calls processListEdit(), which unmarshals the message object into parameters of the add operation, and uses these parameters together with the additional caller name to update the local history and display the input string along with the inputter's name. The received message does not trigger a call to replicatedAdd() to prevent an infinite cycle of adds.

What we have described above is a standard implementation of the replicated architecture [18] except that messages can be routed through a central server if the relayer routing option is used at communicator creation time.

It is possible to use GroupMessages to also implement the centralized architecture [18]. In this architecture, a central master computer stores a master copy of all shared objects, which is typically cached at users' sites to support efficient reading of these objects. Writes to these objects are first made in the central copy and then copied into the cache. Figures 7 and 8 show the GroupMessages implementation of this architecture for the IM application.

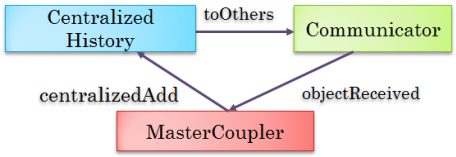

Figure 7 Master IM in Centralized Architecture

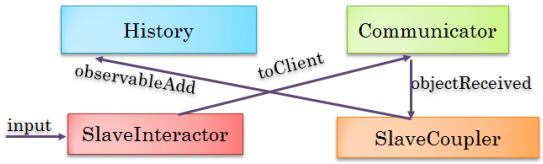

Figure 8 Slave IM in Centralized Architecture

When a slave interactor receives input, it does not send it directly to the local history. Instead, it uses a unicast call, toClient(), to send the new value to the central client. This call takes an additional parameter indicating the client name:

```
void addToHistory(String newValue) {
    communicator.toClient(
      MasterIMModelLauncher.CLIENT_NAME, newValue);
}
```

The master adds the value to the history, and uses toOthers() to send a marshalled message to all slaves:

```
public void centralizedAdd(
       ElementType anInput, String aSourceName) {
   int anIndex = size();
   super.add(anIndex, anInput);
   UserEdit<ElementType> usertEdit = new AUserEdit (
     OperationName.ADD, anIndex, anInput, aSourceName);
    communicator.toOthers(usertEdit);
}
```

As in the replicated architecture, the marshaled message contains the index and value of the add operation. In addition, it contains the name of the inputter, which the slave extracts to determine the output. The reason for sending this name in the

centralized architecture is that the message arrives at the slave from the model, so the message sender parameter provided automatically by GroupMessages does not indicate the (slave) inputter name. In this implementation, each site determines the user-interface. This is the reason for needing the inputter name at each site.

As all sites display the same user-interface, the master could alternatively compute this output, in which case it would not have to send the inputter name. However, it would have to send different outputs to the inputter and other users. To support such communication, GroupMessages offers two additional multicast calls, toCaller() and toNonCallers(). Code invoked in response to a message to client C^1 from client C^2, can invoke these two calls to send message to C^2, and all other clients other than C^1 and C^2, respectively. These two calls are inspired by analogous calls provided by the centralized Suite system. Also motivated by Suite, GroupMessages provides the toClients() call, which takes as an argument an object and a list of client names, and multicasts the object to all clients in the list. Finally, it provides the toAll() call to broadcast a message to all clients, including the one that invoked the call. With these calls, GroupMessages can simulate all multicast groups defined by previous systems.

In the centralized architecture above, all interactors of a slave model are distribution-aware as they communicate with the master model. This problem can be solved by making them make a special proxy add call on the slave model, which can then forward the call to the remote master model. Thus, the distribution-awareness is restricted to only the models.

So far, we have shown how GroupMessages can be used to implement coupling and awareness in centralized and replicated systems. It can also be used to implement control functions. Before allowing a change, a client can check with an access/concurrency control vetoer. Authorization and lock information can be shared in a centralized/replicated architecture using GroupMessages.

To illustrate, let us extend the IM user interface to provide an access control user interface in which the addInputter/addAdministrator commands are used to allow a specific user to provide input and give another user the right to input, respectively (Figure 9).

Figure 10 shows how this functionality can be added to the replicated IM history. A special AccessController object processes the addInputter and addAdminsitrator commands and replicates these operations on all replicas. As the same change is to be made in all replicas, the toAll() call is used.

```
public void replicatedAddInputter(String aNewInputer) {
    String aUserName = communicator.getClientName();
    if (!canAdminister(aUserName))        {
        showNoAdminMessageDialog( aUserName);
        return;
    }
    communicator.toAll(
        new AnInputAuthorization(aNewInputer));
}
```

An extension of ReplicatedHistory, ControlledHistory, checks with the AccessControl object before adding an item.

In this extension, the same session is being used to communicate two kinds of information, the user input and the authorization information, which are processed by different objects, the IMCoupler and AccessReceiver (Figures 6 and 10 respectively). As GroupMessages is unaware of these two subchannels, it passes an incoming message to all listeners (of a specific application session). Thus, each listener must determine, using characteristic of the received object, if it should process the object – a disadvantage of message-based communication. In our example, this task is relatively simple, involving simply the use of the Java **instanceof** operation, as the two receivers process different types of objects. Thus, the access receiver ignores ListEdit objects, as shown below:

```
public void objectReceived
            (Object aMessage, String aSourceName) {
  if (aMessage instanceof AnInputAuthorization)
      processInputAuthorization(Mmessage);
  else if (aMessage instanceof AnAdministratorAuthorization)
      processAdminAuthorization(aMessage);
}}
```

Concurrency control can be similarly implemented by checking and replicating lock information.

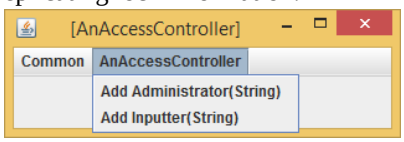

Figure 9 Access Control User Interface

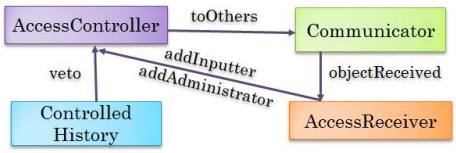

Figure 10 Access Control Architecture

F. Send and Receive Filters

As mentioned earlier, one of our goals was to allow delay, modification, and re-ordering of messages to support programmer-controlled consistency. Apparently, the primitives described so far are sufficient, as we support message-based communication. Instead of forwarding messages to the communicator, message producers can submit them to local consistency modules, which can modify them by, for instance, time stamping them and then delaying them if necessary. Similarly, consistency modules can receive messages, and after reordering, modifying, and/or delaying them, submit them to the actual message consumers.

However, there are several problems with this approach. First, it requires the message producers and consumers to be consistency-aware as they must send (receive) message directly to (from) the communicator or through the consistency modules. Second the consistency modules must implement some of the functions of GroupMessages such as registering of different kinds of listeners and forwarding messages to them. Third, they must delay messages at the sending/receiving sites, which is a non-trivial task. Finally, in relayed communication, centralized consistency algorithms such as Jupiter [19] require

morphing/reordering of messages at the relaying site. The API, described so far, does not provide interception of these messages. To address these problems, GroupMessages provides several additional concepts.

Assuming that consistency module implementers would want to control only the amount of delays and not how the delays are implemented, GroupMessages provides operations that allow programmers to set the minimum and maximum delays to both other clients and the relayer, and given a message directed at a site, delays it by a random value between the two limits for that site. It allows programmer-defined modules to intercept sent messages after they have been submitted to the communicator but before they have been delayed or sent. Similarly, it allows these modules to intercept received message after they have been delayed but before they are distributed to listeners. Finally, it allows programmers to intercept sent and received messages at the relayer. An intercepting module is free to not (immediately) forward a message to the next stage in the communication pipeline and/or modify the message. Such a module is called a filter. The next stage in the pipeline is called a *message processor* and is passed to the filter as a parameter of a filter setter method.

Figure 11 shows the use of send and receive filters to implement causality in the IM application. After a ReplicatedHistory submits a ListEdit to the communicator, the latter (through subcomponents) wraps the edit in a SentMessage and passes this message to the filterMessage() method of the programmer-defined CausalSentMessageFilter. A SentMessage encapsulates not only messages generated by the client through explicit multicast calls but also system-generated messages resulting from client join and leave requests. The filter is given all messages so that it can, for instance, delay all of them using programmer-defined algorithms. This filter checks if the message is a user message, and if so, extracts the wrapped message, time stamps it, replaces the wrapped message with the timestamped edit, and forwards the modified SentMessage to the next sending stage of the communicator:

```
public void filterMessage(SentMessage aSentMessage) {
   if (message.isUserMessage()) {
      message.setUserMessage (
         causalityManager.timeStamp(message));
   sentMessageProcessor.processMessage(
                                 aSentMessage);
}
```

The dual of this event flow occurs at the receiving replica. The timestamped edits are passed to CausalityReceiveFilter, which removes the timestamps (after possibly buffering the messages) and forwards the list edits to the received message processor, which forwards it to the programmer-defined receive listener,IMCoupler we saw earlier. This part of the processing is shown in the trace displayed in the IM console window of Figure 12.

In this trace, Alice, Bob, and Cathy communicate messages directly to each other, and Alice's messages to Cathy are delayed:

```
communicator.setMinimumDelayToPeer(
   CathyP2P.USER_NAME, DELAY_TO_CATHY);
```

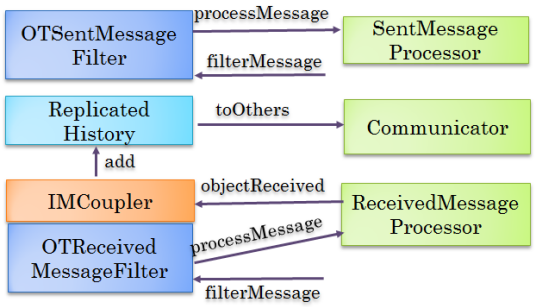

Figure 11 Client-Side Send and Receive Filters

```
I***EvtSrc(MessageGivenToFilter) Msg((Received(Type(ClientJoined), Data()))) Address(Cathy)
I***EvtSrc(MessageGivenToFilter) Msg((Received(Type(NewObject), Data((1, are lovely)
{Alice=1, Cathy=0, Bob=1}))))) Address(Bob)
I***EvtSrc(MessageGivenToFilter) Msg((Received(Type(NewObject), Data((0, The woods)
{Alice=1, Cathy=0, Bob=0}))))) Address(Alice)
I***EvtSrc(ReceivedMessageDistributedToListeners) Msg((0, The woods)) Address(Alice)
The woods(Alice)
I***EvtSrc(ReceivedMessageDistributedToListeners) Msg((1, are lovely)) Address(Bob)
are lovely(Bob)
```

Figure 12 Causal Multicast

Alice enters the string "The woods," in response to which Bob enters "are lovely." Because of the delay, these messages arrive in reverse order at the Cathy's site and are processed in this order by the receive filter. The filters learns from the time stamp of Bob's message that there is an earlier message, so it buffers the message, and when Alice's message arrives, it delivers Alice and Bob's messages, in that order, to the receive message processor, which, in turn gives them to the IM coupler in that order.

V. IMPELEMENTATION AND EXPERIENCE

Message filters expose part of the send and receive pipelines. We briefly outline the other aspects by tracing the flow of a multicast call from a sender to a receiver. The sent message along with the kind of the multicast group to which it is addressed (such as other, all, and caller) are wrapped in a SentMessage data object. This object is then given to the sent message filter, which, as mentioned above, gives it to the message processor. The filtered message then is then put in a sent message bounded buffer, unblocking the caller. The consumer of this buffer is a system-created message-sender thread. At this point the message takes two routes depending on whether it is to be relayed through the central server.

In the case of a relayed message, the message-sender thread computes the delay to the server, sleeps for the required time, and makes an RMI call to the relayer to multicast the message. The relayer passes the message to a central multicaster, which for each destination, wraps the user message along with the name of the source into a ReceivedMessage object, and makes an RMI call at the destination to hand it the message. A separate multicasting thread is created in the relayer for each application session as it is assumed that messages of different applications do not interfere, and thus do not have to be serialized.

In the case of a direct message, the steps are similar except for the following differences. A local rather than central multicaster is used to deliver a ReceivedMessage to each

destination. Moreover, for each destination, a separate thread is created to send messages to the destination, delaying them if necessary. Thus, messages to different destinations can be sent concurrently. We do not create multiple threads for sending messages to the same destination to prevent messages from being delivered out of order. As the underlying IPC layer (RMI) supports synchronous sends, this means that the acknowledgement for a message to a destination must arrive before the next message can be sent. Thus, synchronous IPC conflicts with highly synchronous collaboration (such as sharing telepointer moves), even with (asynchronous) threads.

In both the relayed and direct cases, an RMI call is made at a receiving site to deliver the ReceivedMessage object. These calls put the message in a bounded buffer and unblock the RMI thread. A message-receiver thread is the consumer of this buffer. If a message arrives from the relayer, it calculates the amount of delay to the server, and sleeps for this amount. It then delivers the message to the receive-message filter, which gives it to the receive message processor. The final step is to extract the user data and sender name from the message and pass them as parameters to each receiver listener.

We have used this implementation for creating a variety of student assignments. These include a centralized and replicated implementation of shared Java widgets, and an integrated IM-editor tool that allows users to jointly edit a text area, exchange messages about the editing, and use a telepointer to point at the messages and text area. The assignments involved causality and operation transformation modules for direct and relayed communication, respectively, and jitter filters for reducing jitter in telepointers. None of the previous group communication tools are flexible enough to implement these assignments. Lower-level general purpose distributed computing platforms such as RMI of course offer this flexibility, but programmers would then be responsible for the non-trivial tasks handled by our implementation.

VI. Conclusions and Future Work

This paper motivates a new set of requirements for multicast including support for caller control of message destinations, centralized and replicated architectures, message delay, ordering and change, direct and relayed communication, and multi-application sessions. It identifies features of remote procedure call that conflict with these requirements such as synchronous calls, transparency, concurrent remote invocations, single-site proxies, and remote function calls. It describes a design and implementation of message multicast that meets the requirements.

The design has several new features including two-level sessions, joining a session as a relay client or direct communicator, automatic awareness of the message sender, send and receive filters, and high-level primitives for adding delay and jitter in both direct and relayed communication.

The paper shows how these features can be used to implement (a) centralized and replicated architectures, and (b) coupling, awareness, control and consistency. In all of the examples, the original single-user code was used unmodified, and additional collaboration functions (such as access control and consistency) were added without changing the basic code for coupling users. Thus, while our primitives require collaboration awareness in the application code, different kinds of awareness such as coupling, control, and consistency awareness can be isolated in different modules.

While our driving problem was education and research, there is no reason why our design would not be useful also for building industrial strength applications, which arguably, do not offer more sophisticated synchronous collaboration functions than our running example. Of course, more work is needed to validate our hypothesis – our code is available in a GitHub repository for this validation. Additional research is also needed to integrate our research with the lower-level abstractions supporting RPC and higher-level abstractions supporting collaboration transparency. This paper provides a basis for investigating such support.

Acknowledgment

This research was supported in part by the NSF awards IIS 0810861 and IIS 1250702.

References

[1] Dewan, P., *Tools for Implementing Multiuser User Interfaces.* Trends in Software: Issue on User Interface Software, 1993. **1**: p. 149-172.

[2] Dewan, P., R. Choudhary, and H. Shen, *An Editing-based Characterization of the Design Space of Collaborative Applications.* Journal of Organizational Computing, 1994. **4**(3): p. 219-240.

[3] Roseman, M. and S. Greenberg, *Building Real-Time Groupware with GroupKit, A Groupware Toolkit.* ACM TOCHI, 1996. **3**(1).

[4] Dewan, P. and R. Choudhary, *A High-Level and Flexible Framework for Implementing Multiuser User Interfaces.* ACM TOIS, 1992. **10**(4).

[5] Chung, G. and P. Dewan. *Towards Dynamic Collaboration Architectures*. in *Proc. CSCW*. 2004.

[6] Junuzovic, S. and P. Dewan. *Towards Self-Optimizing Collaborative Systems*. in *Proc. CSCW*. 2012.

[7] Junuzovic, S. and P. Dewan. *Multicasting in Groupware*. in *Proc. IEEE CollaborateCom Conference*. 2007.

[8] Lamport, L., *Time, clocks, and the ordering of events in a distributed system.* CACM, July 1978. **21**(7): p. 558-564.

[9] Sun, C. and C. Ellis. *Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements*. in *Proc. CSCW'98*.

[10] Ellis, C.A. and S.J. Gibbs. *Concurrency Control in Groupware Systems*. in *Proceedings of the ACM SIGMOD '89*.

[11] Knister, M.J. and A. Prakash. *DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors.* in *Proceedings of CSCW'90.*

[12] Krasner, G.E. and S.T. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, August/September 1988 **1**(3): p. 26-49.

[13] Lauer, H.C. and R.M. Needham, *On the Duality of Operating System Structures.* ACM Operating System Review, April 1979. **13**(2): p. 3-19.

[14] Kum, H.-C.M. and P. Dewan, *Supporting Real-Time Collaboration Over Wide Area Networks.* Proc. ACM CSCW 2000.

[15] Stefik, M., D.G. Bobrow, G. Foster, S. Lanning, and D. Tatar, *WYSIWIS Revised: Early Experiences with Multiuser Interfaces.* ACM TOIS, April 1987. **5**(2): p. 147-167.

[16] Dewan, P. and E. Vasilik. *Supporting Objects in a Conventional Operating System*. in *Proceedings of the San Diego Winter '89 Usenix Conference*. February 1989.

[17] Nelson, B.J., *Remote Procedure CallPh.D. Thesis and Tech Report CMU-CS-81-119*. May 1981:

[18] Dewan, P., *Architectures for Collaborative Applications.* Trends in Software: Computer Supported Co-operative Work, 1998. **7**: p. 165-194.

[19] Nichols, D., P. Curtis, M. Dixon, and J. Lamping. *High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System*. in *UIST*. 1995.