S. Bødker, M. Kyng, and K. Schmidt (eds.). *Proceedings of the Sixth European Conference on Computer-Supported Cooperative Work, 12-16 September 1999, Copenhagen, Denmark*

An Experiment in Interoperating Heterogeneous Collaborative Systems

Prasun Dewan

University of North Carolina, USA

Anshu Sharma

Oracle Corp., USA

Abstract: Currently, collaborative systems manipulating the same artifact but implementing different policies and architectures cannot interoperate or "collaborate" with each other. Therefore, it is not possible for users to use different collaborative systems to work on a single shared artifact. As an initial step towards such interoperation, we have carried out an experiment involving the interoperation of two heterogeneous collaborative spreadsheets. The experiment has resulted in some general protocols, techniques, and lessons applicable to the interoperation of systems offering different concurrency-control policies, couplings, and architectures. The paper surveys different approaches along these three dimensions, motivates the rationale for inter-operating them, identifies issues in their interoperation, and presents and evaluates solutions for a small number of interoperation scenarios in the surveyed design space.

Introduction

The promise of computer supported cooperative work has resulted in a proliferation of collaborative systems supporting the same task. For instance, almost every infrastructure developer provides a shared whiteboard; and several custom-ones have been developed as both research and commercial products.

These systems tend to be heterogeneous, that is, offer different policies and architectures for supporting collaboration As the usage of these systems gains popularity, it will become important for them to be able to interoperate with each other so that collaborators can use different systems to manipulate the same artifact. Such interoperation is useful for two related reasons:

- *User Preferences:* Systems tend to be heterogeneous because they make different tradeoffs, each offering a unique set of benefits. Collaborators may wish to use different policies because different features are important to them This is true in single-user interaction For instance, users tend to seldom agree on a particular word-processor or word-processor settings It is not likely to change when they collaborate with others.
- *User Constraints and Training:* Users in different organizations may be trained in and constrained to use different systems. They may not have the time, desire, or ability to learn a new, common system in order to collaborate with each other.

In general, formation of work groups is dynamic, usually in response to an identified need, and brings together people with different training, skills, and work habits. Thus, members of these groups are likely to use different systems

It can be argued that standardization of collaboration systems will eliminate the need for interoperation. Even if standardization is successful in this area, we believe the standardized collaborative tools, like single-user tools of today, will offer a range of parameters Not all collaborators are likely to prefer the same configuration, thereby requiring interoperation at the policy and architecture (but not system) levels. It is these levels that the paper mainly focuses on.

The general concept of interoperation is not new - this is the classical problem of heterogeneous software systems. Specific instances of this problem have been addressed in programming languages, operating systems, and database systems. These solutions are applicable to the collaboration domain, since collaborative systems tend to use different underlying software. However, they are not sufficient for the collaboration domain since they do not address differences in policies and architectures invented in this area.

Because of the newness of this field, researchers and developers have so far concentrated on building new kinds of collaborative systems, disregarding how their tools would work with those offering alternative solutions. The main exception is the work on DistEdit (Knister 90), which explored the use of heterogeneous single-user editors in a single collaborative session. However, we know of no previous work on interoperation of two complete collaboration systems.

In order to understand some of the basic issues and solutions involved in such interoperation, we carried out an experiment involving interoperation of two different collaborative systems. The experiment has resulted in some general protocols, techniques, and lessons applicable to the interoperation of systems

offering different concurrency-control policies, couplings, and architectures. We have found that existing mechanisms for supporting latecomers in a session support interoperation along the coupling dimension. We have also devised general protocols and policies for supporting interoperation along the concurrency-control dimension We have found that the choice of the policies being interoperated influences the fairness of the interoperation, and that tickle-locks and queue-based floor control offer better fairness than regular locks and floor control, respectively.

The rest of the paper is organized as follows. We first survey the dimensions of coupling, concurrency control, and architecture, identifying and evaluating some of the popular solutions along each of these dimensions. The purpose of this section is to give the reader background in some of the concepts needed to understand our work in interoperation, and to show the need for allowing alternative solutions along each of these dimensions to coexist in a collaborative session Next we outline some of the basic issues in interoperation. We then describe the two systems we interoperated, abstracting out details not relevant for the interoperation experiment The next section is the bulk of the paper, describing and evaluating our approach to interoperation of the two systems. Finally, we present conclusions and directions for future work.

Background

Collaboration systems differ along a variety of dimensions. We consider here the dimensions along which our two systems differed: coupling, concurrency control, and architecture. We survey some of the popular solutions along each of these dimensions and enumerate their advantages and disadvantages to show that none of these solutions is preferable over all others.

Coupling

Coupling [Dewan 95] determines how the displays of different users are linked to each other. One of the popular policies for coupling, termed WYSIWIS (Stefik 87) (What You See Is What I See), ensures that all users' displays are identical. This policy is ideal for meetings in which all users share a common focus. However, it has two main problems. First, experience has shown that users get involved in "window wars" and "scroll wars" as they try to place their windows and scrollbars at different positions (Stefik 87). In general, we can expect a war whenever users are forced to share something they do not want to share. Second, broadcasting each event to all workstations creates performance problems, which are particularly unfortunate when users to do not wish to share all events.

Therefore, there has been considerable work in identifying non-WYSIWIS coupling by exploring dimensions along which coupling can be relaxed (Stefik 87, Hill 94, Dewan 95). Many systems allow collaborators to see different views

of a common model. Others also allow them to determine which of the user-interface objects such as cursor positions and scrollbars are coupled. Some support asynchronous coupling, allowing users to determine when changes are transmitted to/received from others. Transmission/receipt of changes can occur explicitly because of execution of special commands or implicitly as a side effect of other commands (such as moving the cursor out of the object changed) or passage of time (e g. coupling every hour). While these non-WYSIWIS schemes give collaborators better performance and flexibility, they have the disadvantage that users do not have "referential transparency", that is, cannot refer to objects based on their screen appearance and location. For instance, "the red square on the upper left corner", may not be meaningful to all users if they can color and scroll their displays independently. Another important disadvantage is that non-WYSIWIS coupling modes cannot be implemented by systems such as Microsoft NetMeeting that allow sharing of existing single-user programs

Thus, each of these coupling schemes has its advantages and disadvantages, and which policy is chosen may depend not only on the collaboration task but also the individual users. For instance, some users may prefer reuse and referential transparency offered by WYSIWIS systems, while others may prefer the performance and flexibility of an asynchronous non-WYSWIS interaction. Interoperation of the heterogeneous coupling schemes would allow the individual user rather than the whole group to make the choice

Coupling Architecture

There is also considerable variation in the architecture used for coupling users. In general, users' actions are processed by multiple application layers such as the kernel, window system, toolkit, and application semantics. Different architectures differ in the way in which they replicate these layers. Some systems completely replicate the application, some completely centralize it, while others centralize the top-level (semantics) layer but replicate the layers below it. These architectures can be described using the generalized zipper model [Dewan 98]. The model assumes that if a layer is replicated, all layers below it are also replicated. Differences in architectures can result from differences in the top-most layer that is replicated. The higher this layer, the more the replication degree of the architecture. This degree is thus increased by "opening the zipper" until we reach the fully replicated architecture. Figure 1 shows the zipper opened to different degrees for the same set of layers.

Replicating a layer has two main advantages. First, it allows objects defined in the layer to be uncoupled, since multiple instances of these objects are created for different users. Second, all processing in that layer is done on the local workstation, thereby giving better performance. The disadvantages are that replicas are difficult to synchronize; not all workstations may have the software and hardware to run a replica, and performing the same operation multiple times

in different replicas can be expensive, cause bottlenecks if a centralized resource is accessed by the operation, and result in undesired semantics if the operation has a side effect (such as printing a check or sending a mail message)

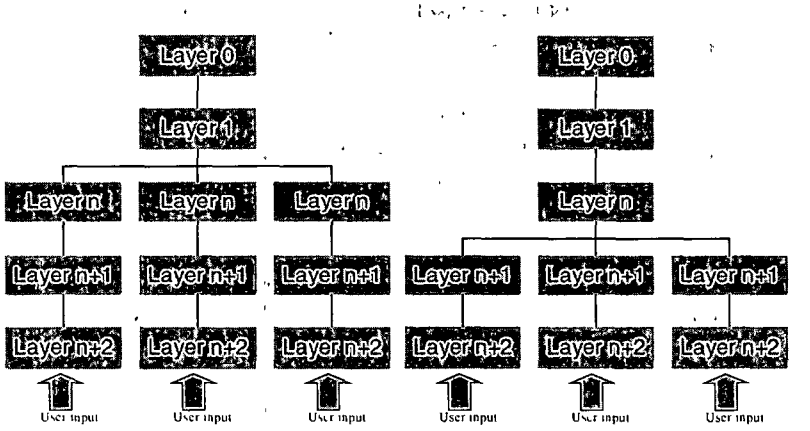

Figure 1: Different Instances of the Zipper Architecture

Thus, in some cases, the architecture of an application depends on its layers In others, it depends on the workstations and couplings of the individual users. In these cases, it is important to interoperate architectures suitable for different sets of collaborators.

Concurrency Control

Since collaborative applications allow multiple users to alter the state of the system, certain inter-leavings of user actions can lead to inconsistent state. Most collaborative systems either provide mechanisms to prevent any inconsistencies to appear in the system, or resolve them at a later (suitable) stage. These mechanisms may or may not involve active user involvement. The term concurrency control is used to describe these mechanisms.

The aim of concurrency control mechanisms is to allow for maximum concurrency while trying to restrict the system from becoming inconsistent or irrecoverable. Different systems differ in the degrees to which they satisfy these two competing goals.

The simplest form of concurrency control is floor control In this scheme, only one user is allowed to give input to the system at any given time. Any user that wishes to become active has to request the *floor*. If the floor is free, the request succeeds and the user gets the floor. If some other user has the floor, the result of the request can be one of many possible options. The simplest scheme is to discard the request. Otherwise, the request may be en-queued, and all the users who have requested the floor may get it on a first-come-first-served basis.

Another option is to have priorities associated with each user or request, and use priority queues. In some variations of this scheme a moderator may decide whom to relinquish the floor to. All these schemes are minor variations of what can be collectively called turn-taking protocols (Ellis 91).

These protocols are easy to implement, and in some cases (e.g mediated meetings) very suitable. However, they limit parallelism since only one user is active at any instant Even if two users' actions do not conflict, they are not allowed to perform them concurrently. For example, two users will not be allowed to edit different cells of a spreadsheet even if there is no conflict. This is overly restrictive for many interactive groupware applications. Moreover, some of the discussed floor control schemes can lead to starvation (e g. priority queues).

These problems are addressed by lock-based concurrency control, which allows users to obtain locks on different objects constituting the collaboration artifact such as cells in a spreadsheet. Two or more users can obtain locks and thus work concurrently as long as they do not wish to work (acquire locks) on the same objects. Locking is a popular concurrency control scheme and has been implemented in several systems such as (Newman-Wolfe 92, Greenberg 94).

Several variations of this basic scheme have been implemented to optimize performance, user-effort, and concurrency. It is usually cumbersome for a user to explicitly request and release locks. In many cases, it is possible for the system to implicitly issue lock requests as a side effect of other operations such as selecting an object (Newman-Wolfe 92, Greenberg 94). At the same time, the user should be allowed to explicitly request locks on several objects to perform atomic operations.

A problem in distributed systems is high latency and, thus, response time. A simple-minded implementation of locking scheme would be to maintain the state of the locks on a single host. In this case, for each lock request, the remote users' host would have to communicate with the centralized process resulting in high latency for remote users. Noting that there is locality of reference in the user's lock requests, some systems cache locks (Prakash 94). When a user no longer needs a lock, instead of releasing the lock to the central lock manager, the lock is locally held until some other user requests the same lock. The requests made by a new user would take more time now because the request has to go to the central manager and then relayed to the host that currently holds the lock. But as long as there is locality of reference and different users work on different subsets of objects, this scheme results in faster response on the average.

If one user holds the lock for a long time, other users may have to wait indefinitely. This problem can be solved to some extent by using the concept of tickle-locks (Greif 86). These locks have an associated time-to-live value associated with them. If the user does not perform any operation during this time, the locks are released automatically. Depending on the application semantics, the value of the object that the user is editing may or may not be committed before

releasing the locks (In some cases, it may be more appropriate to perform an *undo* operation before releasing the locks).

Locking schemes fall into the category of pessimistic concurrency control. A slightly optimistic locking scheme (Greenberg 94) can be implemented as follows to improve the performance Whenever a user requests a lock, the control returns to the application immediately and the user can modify the state of the object while the request is being processed, possibly at a remote site. If the lock request is denied, an undo operation is performed on the object.

Fully optimistic concurrency control (not addressed by our experiment) is offered by systems such as COAST (Schuckman 96) that support transactions A transaction is a sequence of instructions that executes atomically; that is, either all or none of its steps complete execution. Serializability and recoverability are two important properties of transactions. Serializability means that the concurrent execution of a set of actions is equivalent to some serial execution of the same actions. Recoverability means that each action appears to be all-or-nothing: either it executes successfully to completion (*commits*), or it has no effect at all (*aborts*).

Both locking and transactions offer more concurrency control than floor control. An important disadvantage of locking is that it can lead to deadlocks. An important disadvantage of transactions is that they may abort, undoing possibly hours or days of user work Merging can be used as an alternative to abort (Munson 97), but it cannot automatically resolve all conflicts.

Thus, as with coupling and architecture, the choice of the concurrency control may depend on the individual user.

Interoperation Issues

Interoperability is not a new concept. The advent of the internet, which is an interconnection of heterogeneous systems, forced systems designers to create tools such as ftp and mail that would allow users on different systems to work together on shared artifacts. For instance, two users can use ftp to share files that may be stored in a different format on their respective file systems (e.g. AFS and Windows). More recently, CORBA (Vinoski 97) allows objects created on different platforms using different programming languages to interoperate. Java RMI (Remote Method Invocation) allows interoperation among objects executing on different platforms. However, none of the previous systems addresses interoperability along the three dimensions discussed above. Such interoperation requires us to address three kinds of problems.

Semantics

It is not always clear as to what the semantics of two different heterogeneous schemes are when they are integrated. In other words, a consistent state according to one scheme may be inconsistent according to another For example, in

WYSIWIS coupling, all users see the same state at all times, while in a non-WYSIWIS interaction, this is not the case. We will see later how we address this and other forms of inconsistencies that appear in our experiment. To better understand the nature of this problem, consider below a more complicated inconsistency scenario that does not manifest itself in our experiment.

Suppose Jim is using a lock-based system, which means that any changes that Jim makes will be accepted. If Jane wants to use a transaction-based concurrency control scheme, then it is not clear as to what should happen in case of conflicts Suppose Jim acquires a lock on cell A with value "Apple". Now, Jane reads the value of the cell A ("Apple") and wants to change it to "Orange". Since, Jane read a correct value and changed it to another correct value, according to transaction semantics, the change will be committed But this is wrong according to the lock-based system's consistency criterion, since as long as a user has a lock on an object, no other user can change the value. Now, if Jim writes a new value ("Banana") into cell A, his changes should be aborted by transaction semantics. But there is no concept of abort in lock-based systems

A solution to this problem would be to deny any transaction operation as long as any user has a lock on the object However, in a collaborative application it is not very clear as to what constitutes a read operation, since users may not explicitly execute this operation. Typically, the system responds to the commitment of a change by one user by updating the displays of the remote users without knowing if a remote user has actually read the value. A conservative approach would be to assume that all values displayed to the user are considered as read, but this would unduly limit the concurrency.

Implementation

In general, groupware applications and infrastructures are built on different platforms and do not adhere to any standard. For instance, Habanero (Chabert 98) is a Java-based system while Suite (Dewan 95) uses C and has its own RPC-based communication mechanisms. In order to interoperate the systems, both the semantic and syntactic gaps have to be bridged. A language- independent system such as CORBA can be used to allow objects on different systems to communicate with each other. Thus, the problem is reduced from providing many-to-many translations to providing a translation between every system and a standard such as CORBA. These problems have been studied and solutions suggested and implemented elsewhere. In our solutions, we assume that the systems are implemented in Java or have proxy objects written in Java The proxy objects may communicate with the existing systems through any suitable mechanism such as CORBA.

Deadlock, Fairness

These two issues arise in interoperation of concurrency control schemes Apart

from being correct, these schemes should also be fair and not susceptible to deadlocks. As we saw earlier, different schemes fare differently on these criteria. Floor control with queuing is fair and precludes deadlocks. On the other hand, lock-based concurrency control can lead to starvation and deadlocks. When interoperating schemes with different behaviors with respect to fairness and deadlocks, it is difficult to ensure that the resulting system does not introduce new forms of unfairness and deadlocks that were not present in the original systems. The design of interoperability mechanisms should take these factors into consideration.

Case Studies

Before we discuss how we addressed these issues in our interoperability experiment, we describe the two systems we interoperated.

Habanero Spreadsheet

Habanero (Chabert 98) is a system developed at NCSA that provides a toolkit for building collaborative applications. It supports full replication by running a copy of the application at each user's machine. It receives events from each replica and broadcasts them to all other replicas. It also supports latecomers by dynamically creating a new replica for the new user. Finally, it provides flexible mechanisms for implementing coupling and concurrency control. It provides a central module for sequencing the coupled events received from different replicas.

We used Habanero to implement a simple spreadsheet offering fixed synchronous, near-WYSIWIS coupling (that allows users to scroll independently) and ordinary and queue-based based floor control

UNC Spreadsheet

We also implemented, using libraries developed at UNC, another version of the spreadsheet that differed from the Habanero spreadsheet in the architecture, coupling, and concurrency-control dimensions. It is built using the model-view-controller paradigm (Krasner 88), with the controller combined with the view. Instead of replicating the whole application, it replicates the view but not the model. It offers flexible non-WYSIWIS coupling, allowing users to choose whether they wish to send values incrementally (as they type in a cell), or when they move the cursor away from the cell. It offers a large range of locking policies including centralized, cached, optimistic, and tickle locks. Like the Habanero spreadsheet, this spreadsheet supports latecomers by dynamically creating a view for the new user, and allows independent scrolling.

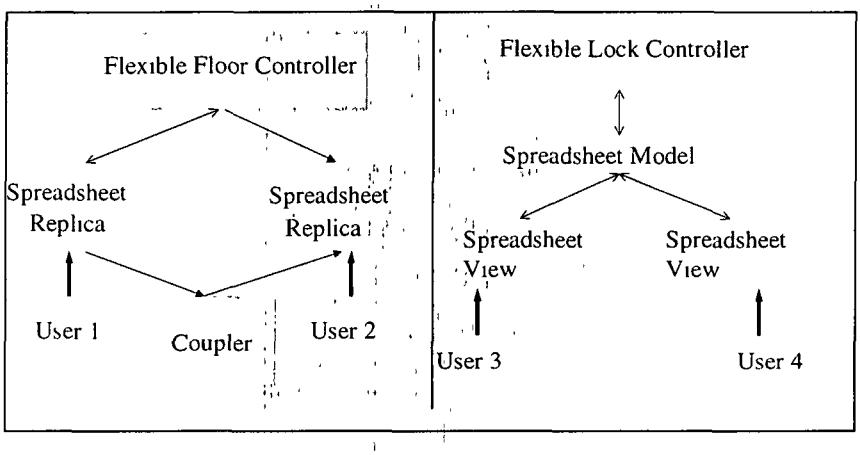

Figure 2 Habanero Spreadsheet (left) and UNC Spreadsheet (right)

Interoperation Policies and Mechanisms

Given two systems used by two different sets of users, we must consider two aspects of the behavior of the interoperating system.

- Local Behavior: The behavior of each system with respect to its users. Thus, in our experiment, it defines the architecture formed for each set of users, and the semantics of coupling and concurrency control for each set of users.
- Interoperation Behavior: The behavior of the system with respect to users of different systems. Thus, in our example, it defines how the architecture of each system is changed to accommodate the other users, and the semantics of the coupling and concurrency control between users of the two different systems.

Ideally, the local behavior of the original system should not change in the interoperating system. Otherwise, the added flexibility of collaborating with the other set of users comes at the cost of changes to the preferred mode of collaboration among users of each set. Therefore, in our interoperation experiment, we ensure that the local behaviors along each dimension are preserved.

Ideally, also, the interoperation behavior should be the same as the local behavior of each system to allow each set of users to use the preferred mode of collaboration with the other set. Of course, this is not possible when the two behaviors are different Thus, an interoperation behavior must be defined for each dimension of heterogeneity that is consistent with both local behaviors (by being an abstraction of them) and as close to them as possible.

Coupling

Let us first consider the dimension of coupling. The coupling semantics of the two systems to be interoperated are:

- Habanero Spreadsheet: Each user sees the same values as the other user. Each change to a value is immediately sent to other users.
- UNC Spreadsheet: Users changes to shared objects can be buffered, and these changes are seen by other users when they are committed (implicitly or explicitly). Coupling settings determine whether changes are committed on each keystroke or when the user moves the input cursor away from a cell.

The interoperating semantics cannot be the same as both of the local semantics, which are different. Therefore, we define the following compromise semantics for interoperation: Changes made by one user to an object are seen by other users sharing the object when these changes are committed (implicitly or explicitly).

These semantics meets our requirement of being an abstraction of the two local semantics. In fact, they seem identical to the coupling semantics of the UNC spreadsheet. However, there is a subtle difference. In the case of the UNC spreadsheet, each user knows that others can buffer changes, and thus hide intermediate results that are not to be discussed. Thus, they may think it rude if they receive such results. However, in the interoperating spreadsheet, this assumption cannot be made regarding users of the Habanero spreadsheet. Thus, users of the UNC spreadsheet must treat local and remote users differently. Users of the Habanero spreadsheet must, of course, treat local and remote users differently, since they are guaranteed synchronous coupling with the former but not the other. These semantics are realized in our example by keeping the model of the UNC spreadsheet consistent with replicas of the Habanero spreadsheet.

Coupling Architecture

Figure 3 shows the interoperation architecture for implementing these coupling semantics. We extended the Habanero spreadsheet so that it appears to the model of the UNC spreadsheet as a view. When users of the two systems need to join each other in a collaborative session, we dynamically add this modified spreadsheet as a new user to both systems. Since it does not interact with an actual user, we refer to its "user" as "dummy user".

The modified spreadsheet translates between the events received from the two systems to ensure that the UNC model is kept consistent with the Habanero replicas. The UNC model and the Spreadsheet replicas expect to receive new cell values (from view and other Habanero replicas, respectively) as strings and CellValue objects, respectively. A CellValue object is basically a wrapper of the cell string entered by the user; therefore, the translation between these two kinds of values is straightforward and simply involves wrapping and unwrapping a string. However, the architecture of the system remains the same even for more complex translation logic. For instance, if the Habanero spreadsheet had

implemented full WYSIWIS interaction by coupling the scrollbars, the translator would simply ignore these events. Similarly, if the UNC spreadsheet had stored cell values as integers, the translator would be responsible for parsing and unparsing the string and integer representations, respectively

The modifications to the Habanero spreadsheet were minor and consisted of about 100 lines of code. The modified spreadsheet implements only those aspects of the view of the UNC spreadsheet that are relevant to the model. It does not implement other aspects (such as changing the coupling policy) that are seen by the user but not the model. As it turns out, it implements the full functionality of the Habanero spreadsheet. This is because we did not find a convenient way to execute different programs for the different replicas of the Habanero spreadsheet This means each Habanero replica must implement the union of the functionality needed by both the real users and the dummy user. The UNC spreadsheet allows the views of a model to execute different programs. Otherwise, the program executed by the dummy user would have had to implement the union of the functionality of the two systems and the translation function.

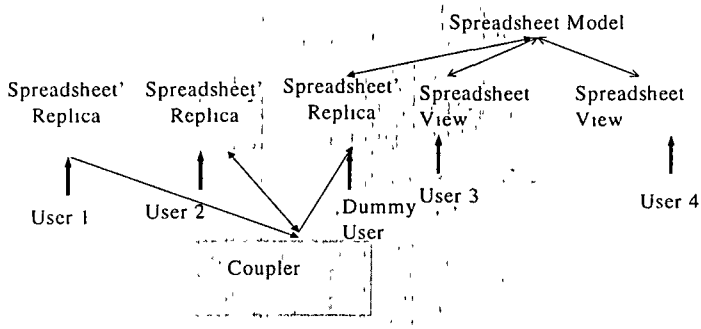

Figure 3 Architecture for Interoperating Coupling

We can describe the approach used in our interoperation architecture in terms of the abstract zipper model described before (Figure 4) Two different architectures can be made to interoperate by creating a module that appears to be a branch of both architectures and translates between the events received from the two architectures according to the coupling interoperation semantics.

If both systems support latecomers, then the new branch can be created dynamically to allow the two groups of users to join each other after they have done some intra-group collaboration. If the two systems also allow the branches to execute different programs, then the individual systems do not have to be changed. The users of each system appear to the users of the other system as a single user.

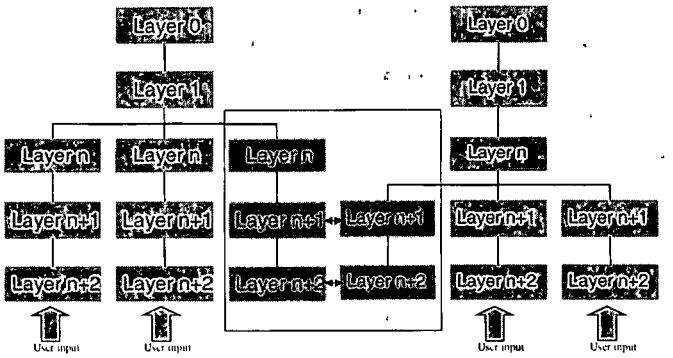

Figure 4 Interoperation in Terms of the Zipper Model

Concurrency Control Semantics

Let us now consider the dimension of concurrency control. Its semantics in the two systems are:

- Habanero Spreadsheet. The default is ordinary floor control. Users can change it to queue-based floor control.
- UNC Spreadsheet: The default is ordinary (implicit and explicit) centralized locks Users can change it to tickle, optimistic, or cached locks

We use the following interoperation semantics: If the user has reserved an object (through floor control or fine-grained locking), then no other user can manipulate the object.

As in the case of coupling, the interoperation semantics are a weaker form of the semantics of the two systems. As we shall see later, they are weaker than we might desire, since they make no promises about fairness. For certain concurrency control modes (queue-based floor control and tickle locks), we will be able to offer stronger semantics with better fairness properties.

Concurrency-Control Architectures

We implemented two different architectures for interoperating the concurrency control of the two systems. In one, the floor-controller is the master or server, and the lock-controller is the slave or client (Figure 5); while in the other, the reverse is true. The server system is the one that keeps global information about which objects are reserved (through locking or floor control). The client system must check with it before granting a reservation request to its users. The client system makes this check in addition to the check it makes for local conflicts. All users of the client system appear to the server system as a single "dummy user".

There is a third architecture possible in which both systems are considered equal and replicate the reservation information. However, we did not consider this

alternative.

Unlike coupling, our schemes for interoperating concurrency-control require the two systems to follow certain protocols that collaboration systems currently do not. As we will argue later, these protocols also support extendibility, and are thus useful even if interoperability is not a goal. Nonetheless, since current systems must be changed to implement these protocols, we may be constrained by what kind of changes can be made easily to a system. The architecture determines the protocol – thus, which architecture is chosen depends on what kind of changes can be easily made to a system. It is for this reason we defined both architectures and associated protocols to give the interoperator some flexibility.

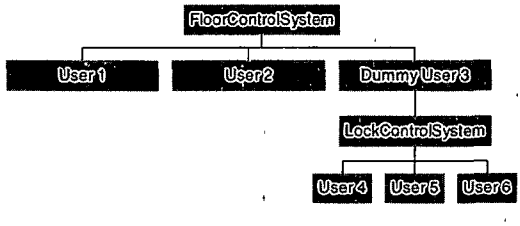

Figure 5 Floor Controller as Master

General Protocol

We first present a general protocol that is sufficient to implement both architectures, and later, under the discussion of each architecture, identify which aspects of this protocol are necessary for it.

We assume that the floor-control system generates the following events

- *FloorRequestedEvent*: This event is generated whenever a request for the floor is made by the user. This is a *vetoable* event, which means that any of the event listeners can veto the event, and the object that generated the event gets notified of the veto. We assume that whenever the *FloorRequestedEvent* is vetoed, the floor-control system denies the request. The event also contains reference to a *principal* object that identifies the user.
- *FloorAcquiredEvent*: This event is generated by the floor-controller when a user successfully acquires the floor. This event also contains reference to the principal. This event is necessary because one of the (possibly) several event listeners may have vetoed the request for the floor.
- *FloorReleasedEvent*: This event is generated when the floor is released by a user and also contains a reference to the principal.

We also assume that the floor-controller acts as a server for the operations, *AcquireFloor* and *ReleaseFloor*, which take a principal as an argument.

Similarly, we assume that the lock-based concurrency control system generates

the events *LockRequestedEvent*, *LockAcquiredEvent*, *LockReleasedEvent*, and serves the requests, *AcquireLock* and *ReleaseLock*, which are like the corresponding floor-control events/requests except that they take an extra *lockable* argument.

Though this protocol has been defined for supporting interoperability, it also supports extendibility, since it allows user-supplied modules to veto events and make requests.

This protocol does not take into account differences in the various forms of the locking and floor-control policies implemented in the system, abstracting out these differences Thus, the interoperation code we give below will work for all variations of these two policies implemented in the two systems, though, as we will see below, the fairness of the implementation will depend on the exact variation.

Neither of our two systems originally implemented this protocol Implementing the complete protocol required about 10 lines of additional code in each spreadsheet

Floor Controller as Master: Policy 1

Figure 6 illustrate the run-time architecture of the interoperability mechanism of this policy. The LockToFloorProxy is the representative of the Lock-Controller to the Floor-Controller, translating between the lock events and the floor requests.

The proxy executes the following pseudo algorithm:

```
LockToFloorProxy

//Lock system wants a lock
On (LockRequestedEvent(Lockable item, Principal user)) do
        // check with floor controller if dummy user can get lock
        If ( ¬ floorControllerObject acquireFloor( LockUser ) )
                Veto
// Keep track of which items are locked so that floor can be released
On (LockAcquiredEvent(Lockable item, Principal user)) do
        Lock-status[item.id]=LOCKED,
//When all locks are released, release floor
On (LockReleasedEvent (Lockable item, Principal user)) do
        Lock-status[item]=FREE,
        For all lockables item do
                If (Lock-status[item]==LOCKED) Return,
        floorControllerObject.releaseFloor( LockUser )
```

It implements the interoperation semantics discussed earlier. When a lock user requests a lock, the lock-system first checks for local conflicts and then asks the proxy to acquire the floor from the server as the dummy user, *LockUser* Thus, locks are denied as long as the floor is not with *LockUser*. Conversely, when any user has a lock on an item, the floor is with *LockUser* and thus only the users of the lock-based system are active. And since the lock-based and floor-control systems are (assumed to be) correct, the resulting system also correctly

implements the interoperation semantics In order to release the floor when all locks are released, the proxy must duplicate the semantics of the lock controller by keeping track of which items are locked

In this policy, the lock-based system would have an unfair advantage, as even if one item were locked by any user, the floor would stay with the lock-based system. So, the floor-control users may be easily starved. It is important to note here that even though the floor-control and lock-based systems may be free of starvation, the interoperation scheme introduces starvation into the system The starvation is due to the fact that *LockUser* can hold the floor for infinite time even if the users of the lock-based system acquire the locks for finite intervals

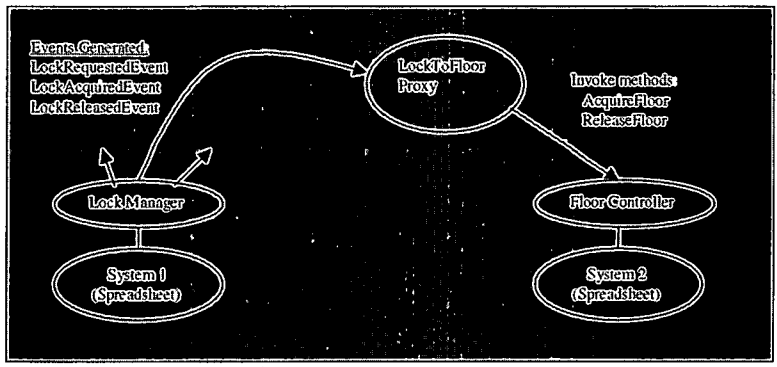

Figure 6 Proxy-based Runtime Architecture

Such starvation is particularly unfortunate when the floor-control system enqueues requests, since if the floor is with the lock-based system, the users of this system can continue to lock new items even after a user of the floor-control system makes a request for the floor.

Floor Controller as Master: Policy 2

This policy fixes the problem above and assumes queue-based floor control We add two additional objects to the runtime architecture – a *FloorListener* object, and a *State* object. The FloorListener listens for events from the floor controller in order to determine if there is a pending request for the floor. This information is now used by LockToFloorProxy to determine if future lock requests should be denied. It is stored in the State object, which is shared by FloorListener and LockToFloorProxy. The code below describes the behavior of the three objects.

This code differs from the previous one in that whenever a user requests the floor, a flag is set, and any subsequent lock requests are denied. The flag is reset whenever a user of the floor-controller gets the floor. We could instead use a count of the number of users in the floor queue, but that could starve the users of

the locking system, since the count may never go to zero.

```
State

Boolean floorRequested initially false

LockToFloorProxy

On (LockRequestedEvent(Lockable item, Principal user)) do
        // floor system has requested floor, disallow further locks
        If( floorRequested ) Veto,
        If ( NOT floorControllerObject acquireFloor( LockUser ) )
                Veto

On (LockAcquiredEvent(Lockable item, Principal user)) do
        Lock-status[item id]=LOCKED

On (LockReleasedEvent (Lockable item, Principal user)) do
        Lock-status[item.id]=FREE,
        For all lockables item do
                If (Lock-status[item id]==LOCKED),
                        Return,
        floorControllerObject releaseFloor( LockUser )

FloorListener
// keep track of whether floor-control system has a pending request
On (FloorRequestedEvent(Lockable item, Principal user)) do
        floorRequested=true,

On (FloorAcquiredEvent(Lockable item, Principal user)) do
        floorRequested=false,
```

In comparison to the previous policy, this is fairer to the users of the floor-control system. Assuming that users of either system reserve the floor/object for finite times, users will be able to get their reservations in finite time. This time can be further decreased (for both systems) if tickle locks are used.

Lock Manager as Master

Finally, we consider the implementation of the second policy under the dual architecture that assumes the lock-manager as the master. The LockToFloorProxy and FloorListener objects are now replaced by their duals, a FloorToLockProxy and a LockListener object. A floor request from the slave is mapped by the proxy to a reservation of all the locks in the master. The floor release event asks the lock system to release all locks

As before, a shared state object keeps track of a pending request from the floor-control system to prevent new locks from being acquired by users of the lock system. Policy 1 can be implemented for this architecture by ignoring such a request. As can be seen by the code, the two architectures and associated policies require different aspects of the general protocol to be implemented The first architecture requires the floor-control system to implement the server operations for acquiring and releasing the floor, while the second architecture requires the

lock control system to implement the server operations for acquiring and releasing locks. The first policy, which ignores queued floor requests, does not require the server system to broadcast events, while the second policy does.

```
State
Boolean floorRequested initially false

FloorToLockProxy

//Get all locks or none to prevent deadlocks
On (FloorRequestedEvent(Principal user)) do
        For all lockables item do
                If( NOT acquireLock(item, FloorUser) Veto,
                        For all lockables item do
                                ReleaseLock(item, FloorUser),
                        floorRequested = TRUE,

On (FloorAcquiredEvent(Lockable item, Principal user)) do
        FloorRequested = FALSE,

On(FloorReleasedEvent(Lockable item, Principal user)) do
        For all lockables item do
                ReleaseLock(item, FloorUser)

LockListener

// Disallow further lock requests if there is a pending floor request
On (LockRequestedEvent(Lockable item, Principal user)) do
        If (floorRequested) Veto,
```

It is important to ensure that the interoperating system not only not introduce inter-system unfairness but also deadlocks. In all of our policies, the two systems cannot simultaneously reserve shared items. When the floor-control system is the master, this is easy to ensure, since either the lock-system or the floor-system has the floor. When the lock-control system is the master, we make sure that if a floor request cannot get all locks, it releases all locks it does manage to acquire

In our implementations, as mentioned before, the entire concurrency control protocol has been implemented by both systems. This has allowed us to dynamically change the architecture and policy at runtime.

The proxy and state code required to implement all of the policies above is about 800 lines. Recall that this code resides outside the individual systems, and interacts with these systems using the concurrency-control protocol.

Conclusions and Future Work

The surprising result from this work is that it is possible to interoperate a synchronously-coupled, fully replicated, floor-control system with a flexibly-coupled, partially centralized, lock system; that it is possible to reason about the semantics of the interoperating system; that it is possible to devise clean architectures for interoperation; and that such few changes are required in the two systems to interoperate.

In this paper, we have motivated the need for interoperating the two systems

by showing that the features they implement are found in a large number of existing systems and that each set of features has important advantages that the other set does not have. We have identified interoperation semantics that are abstractions of and close to the local semantics of the individual systems.

We have shown that existing latecomer support can be used to implement the interoperation semantics for coupling, and that if the existing systems allow different branches in the zipper architecture to execute different programs, then interoperating the coupling does not require changes to the two systems.

We have identified two different proxy-based architectures for interoperating the concurrency-control components of the two systems. In one, the floor-control system keeps global reservation information, acting as a server to the lock system; while in the other, the reverse is true. For each architecture, we identified two interoperation policies. The first policy applies to both ordinary and queue-based floor control, but can result in starvation even if the original systems are free of starvation. The second policy is fairer but applies only to queue-based floor control.

These architectures and policies require the interoperating systems to follow a certain concurrency-control protocol that current collaborative systems do not. The exact protocol depends on the architecture and policy used. We have identified a general protocol that applies to both architectures and policies. It does not take much code to implement and is also useful for extendibility.

The question that this paper does not answer is whether users would be satisfied with interoperation semantics that are different from their preferred semantics. Given that the alternative is either not collaborating with users with different preferences or conforming to a standard system and policies, we believe the answer is yes. A firm answer to this question requires modifications to production software systems and extensive user-studies, which are beyond the scope of this work.

Future work is also required to address interoperation policies for concurrency control with better fairness properties. Moreover, it will be useful to tackle interoperation issues not addressed by this work such as interoperation of more than two systems; and interoperation of other points in the large design space of collaboration systems. In particular, it will be useful to study how popular awareness policies can be interoperated. The approach used in our work of reducing all users of a foreign system to one user is not likely to give the desired awareness interoperation-semantics, and adaptations/extensions to it will be required. Finally, it is important to determine the limits of interoperability – when are two competing approaches so inconsistent that no useful interoperation semantics can be defined. This paper provides a framework for attacking some of these unresolved issues.