Generic and Composable Latecomer Accommodation Service for Centralized Shared Systems

Goopeel Chung, Prasun Dewan and Sadagopan Rajaram
Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175, {chungg,dewan,rajaram}@cs.unc.edu

Key words: multi-user interface, collaboration system, logging, groupware, latecomer, newcomer, window system, genericity, composability

Abstract: It is important that a shared application allow a latecomer to join other users who are already working together with the application. We have developed a latecomer accommodation service framework for centralized shared systems (applications and infrastructures). It employs an independent latecomer accommodation server that is dynamically composable with its clients. The server, also called the logger, logs a shared application's user interface (UI) changes in response to calls made by the client, also called the loggable. Later, when the time comes to accommodate a latecomer, the logger replays the logged changes to the loggable, which, in turn, creates the latecomer's user interface. To deal with UI protocols at different levels of abstraction, we have defined the API in terms of a generic UI model. This reduces the burden on a loggable from a complete service implementation to a translation between its specific UI protocol and our generic UI model. To reduce the space and time overhead, the logger performs complex log compression. The extent of compression depends on the amount of semantic knowledge that the loggable provides to the logger. In this paper, we motivate, describe and illustrate the approach, and outline how it is implemented.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35349-4_22
S. Chatty et al. (eds.), *Engineering for Human-Computer Interaction*
© IFIP International Federation for Information Processing 1999

1. INTRODUCTION

The composition of users participating in a collaboration session with a shared application can change dynamically. A collaboration session can begin with some number of users, and later, some additional users may late-join the collaboration session, and some users may leave early. Moreover, users may not be able to finish the collaboration within one session - they may have to stop the session temporarily, and resume later at another session using the previous session state. Latecomer accommodation service allows latecomers/resumers to join/resume a collaboration session that has already made some progress. We will use the term, latecomer, to also mean resumer.

The essential part of the latecomer accommodation service is to create a new user interface for the latecomer that shows some part of the shared application state. How this is accomplished can vary. The exact part of the state shown may depend on the latecomer's role in the collaboration. Moreover, before displaying the state, the latecomer accommodation service may also show a quick animation of how the user interface reached its current state.

In general, implementing a latecomer accommodation service is difficult, and hence, should ideally be done by a collaboration infrastructure. This is the approach taken in many systems such as Suite (Dewan, Choudhary, 1992), XTV (Chung, Jeffay, Abdel-Wahab, 1994), a system by (Manohar, Prakash, 1995), and Habanero (NCSA). However, each of these systems provides its own latecomer accommodation service. Thus, there is no code sharing among these systems. Moreover, the latecomer accommodation service is tightly integrated with other aspects of the system. It is not possible for applications to use the latecomer service without using other protocols offered by the system such as those for user interface, concurrency control, and access control. It is often the case that these protocols are too rigid for an application.

Therefore, it would be useful to provide an independent latecomer accommodation service that can be used by a variety of systems and applications. In this paper, we present our first step towards such a service. It has the following distinguishing features.

- Composability: We provide a separate server module that is dynamically composable with a latecomer client (a system or an application). The latecomer server can share the same address space with the client, or be in a different address space of the same host, or on a different host. It exports an API that separates the latecomer accommodation service from other collaboration functions. Through the API, the latecomer client sends the UI state change information as often as necessary, and the latecomer server logs this information. When the time comes to accommodate a latecomer, the latecomer server replays the log back to the latecomer client. The latecomer client, in turn, can use the replayed log to create the latecomer's user interface. Our log and replay approach implies "You Get What You Log" - the latecomer client determines what actions are logged and replayed. Since we are using a log and replay approach, we will refer to the latecomer server as the logger, and the latecomer client as the loggable.

- Genericity: Different loggables can provide UI change information based on different UI protocols, which can be at different levels of abstraction. In order to support various protocols, the logger's API is based on a generic UI model that makes few assumptions about how a shared application makes changes to its user interface.
- Log Compression: Our log and replay approach allows animation of how the user interface reached its current state before the latecomer joins the session. However, it may be the case that the latecomer does not want such animation, but simply wants to see the current UI state as quickly as possible. We provide special support for such a situation by compressing the log so that not all of the UI changes sent by the loggable are actually logged. The extent of the compression performed by the logger is proportional to the amount of semantic information given to it by the loggable.
- Ease of Programming: The service should not require a significant burden on the loggable. Instead of implementing the latecomer service, the loggable takes on a translation role, simply converting between its specific UI protocol and our generic model. Such a translation module can be built for a whole class of systems and applications, and we believe that the effort required to build such a module is small.

The rest of the paper is organized as follows. We first describe the related work on which our research is based. Next, we describe our approach. Finally, we give conclusions and directions for future work.

2. RELATED WORK

Many collaboration systems such as Suite (Dewan, Choudhary, 1992), XTV (Chung, Jeffay, Abdel-Wahab, 1994), GroupKit (Roseman, Greenberg, 1996), a system by (Manohar, Prakash, 1995), and Habanero (NCSA) support latecomer accommodation using different approaches.

Suite is a collaboration infrastructure that provides to its application programmers the high-level abstraction of a shared active variable. Suite determines how an active variable is displayed based on its type. The Suite architecture is based on the model-view framework - where a central model implements the shared application state, and multiple views implement interfaces of different users. Suite allows different users' views to be different, thereby supporting non-WYSIWIS interaction. It also supports transactions, where not every change made to a view is immediately committed in the model. When a latecomer joins a collaboration session, Suite calls a load method on the model, which sends to the latecomer's view data structures that have been committed by pre-existing views. Thus, the new user cannot see any uncommitted values.

XTV is an extension of a single-user window system called X that allows users to share existing single-user X applications. The X window system provides to its application programmers the abstraction of a window, which is much lower level

than Suite's active variable. An X application sends to an X server a series of requests to create and update windows, and the X server sends to the X application user input in the form of low-level mouse and keyboard events. XTV enables sharing of a single-user X application by intercepting and distributing the X requests to multiple X servers, and by relaying the user input events from the multiple servers to the shared application. The latecomer accommodation service in XTV works by logging the low-level window requests from the shared application, and replaying them to the latecomer's X server. XTV has to use the logging approach since it is not possible to look inside existing X applications for internal data structures. This is different from Suite, which has its application export the data structures. XTV also performs log compression by maintaining only the current UI state. However, the log compression algorithm is closely tied to the X protocol.

These two systems assume a single centralized component that represents the shared application state, and multiple UI components for different users. Some collaboration systems replicate the shared application component as well as creating multiple UI components. A replicated system provides good response times to its users, because all of their input is locally processed. However, it must perform an additional step of replicating the shared application component for the latecomer. The replicated approach is taken by many systems such as GroupKit, a system by (Manohar, Prakash, 1995), and Habanero, and we will look at those systems now.

GroupKit is a toolkit that lets developers build groupware applications. In GroupKit, when a latecomer joins a collaboration session, the run-time infrastructure sends an event called updateEntrant to an existing application replica, which, in turn, communicates with the latecomer's replica to update the latecomer's user interface. What specific actions are taken during this time totally depends on the shared application. For example, a group drawing program could send the entire drawing to the latecomer, and a text chatting program could send the text contents of all the existing chat windows. Therefore, the latecomer accommodation service is flexible in that it can be whatever the application needs. However, it has to be implemented manually by the application programmer.

A system by (Manohar, Prakash, 1995) allows flexible support for resuming a collaboration session by using a data artifact called a session object. A session object captures a user's interaction with an application, and records window events such as mouse and keyboard actions. When a latecomer resumes the collaboration session, the system replays the session object's window events to a fresh copy of the application, and, in this process, the application creates the user interface for the latecomer. During replay, the system shows the previous user's actual interaction as faithfully as possible by controlling the rate of replay. The system also provides a VCR-like user interface, so that the latecomer can pause, skip, and fasten the replay. In order to support skipping to an upcoming portion of replay, the session object also has intermittent state checkpoints where the previous application wrote its entire current state information. The new application can use one of these checkpoints to load the entire application state without replaying prior window events. The system reduces the burden on the application programmer since history replay is managed by it. This is unlike GroupKit, where an application has to be totally responsible for latecomer accommodation. This is more like the XTV approach, which also logs and replays window system level protocols. But it is different from XTV in the

following respects. First, it works for a replicated system as mentioned before. Second, it allows UI change animation, and allows users to control its replay, while XTV simply creates the current UI state. Third, unlike XTV, it does not compress events.

Habanero is a collaboration system that allows users to interact with a shared application called a hablet. A hablet is a Java applet program extended for multi-user collaboration. When a latecomer arrives, Habanero calls a marshallSelf method that is to be implemented by the hablet. In response to this call, the hablet marshalls itself (i.e. saves its state information) to a marshall object. Habanero, then, migrates this marshall object along with the hablet code to the latecomer's workstation, where the migrated hablet code unmarshalls the previous hablet state from the marshall object. In addition to updating its state, the migrated hablet has to update the latecomer's user interface. Habanero also supports session record and replay feature, but this is a separate mechanism that is automatically supported by the system. Therefore, a hablet programmer is to be concerned only about marshalling and unmarshalling the hablet state when it comes to latecomer accommodation.

As we can see, none of these systems (both centralized and replicated) has a separate module for latecomer accommodation service; the latecomer accommodation service is closely integrated with the system or individually implemented by each application. Moreover, an application that wants to use the latecomer accommodation service from one of these systems must follow all the protocols offered by the system, including those not relevant to latecomer accommodation such as the active variable abstraction of Suite, the low level X protocol of XTV, the groupware programming abstractions of GroupKit, the window events protocol of the system by (Manohar, Prakash, 1995), and the Java applet "habanerization" of Habanero. In the next section, we will describe how these limitations can be overcome for a centralized system. We leave handling latecomers in a replicated system as future work.

3. APPROACH

3.1 Architecture

The architecture that should be formed for our latecomer accommodation service to work is illustrated in *Figure 1.* This architecture assumes that a shared system consists of an application client and multiple UI servers, one for each user. Each server manages the interface of a user for the client. The client sends requests to change the user interface in terms of some UI abstraction the server defines, such as a window or an active variable. The client can also ask the server to notify the client

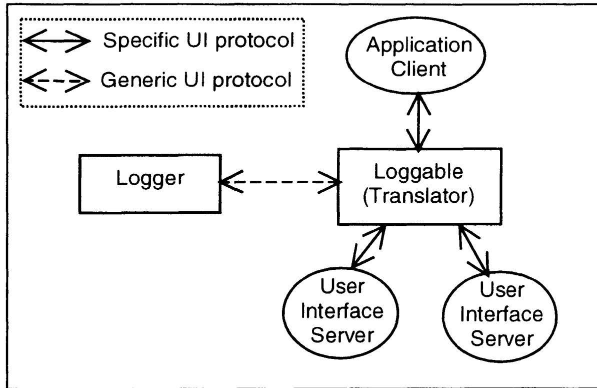


Figure 1. Latecomer Accommodation Service Architecture.

through events when certain aspects of the user interface are changed by the user (for example, mouse movements or active variable changes). Therefore, in a shared system, a client and multiple servers send messages (requests and events) back and forth following a specific UI protocol based on the UI abstraction.

A logger and a loggable work together to provide latecomer accommodation service for the shared system. The logger is a separate, log and replay module. The loggable is positioned between the client and the servers, and listens to messages exchanged between them. The logger and the loggable talk to each other using a generic UI protocol to which a specific UI protocol can be translated.

The loggable translates requests from the client into abstract primitives defined by our generic model, and sends them to the logger, which, in turn, logs them. The loggable has the discretion to decide which request it translates and sends to the logger. When a latecomer needs to be accommodated, the loggable asks the logger to replay the abstract primitives that are needed to create the latecomer's user interface. The logger, in response, sends these primitives. When the loggable receives a primitive, it reverse-translates the abstract primitive to a request of the specific UI protocol. Once the translation is done, the loggable sends the request to the latecomer's UI server.

The major factor in determining the efficiency of our latecomer accommodation service module is the compression ratio of the actual log size to the total size of all the primitives sent by the loggable. The compression ratio is greatly affected by how much semantic information the loggable can provide about the client requests. We roughly classify the extent of information into three cases, and provide an approach for each case. We will describe the three approaches in the sections that follow. Each approach assumes a different model of how the client makes changes to the user interface depending on the amount of semantic information that it can get from the loggable. Each approach will be a generalization of the previous approach, and assumes more semantic knowledge. We could have described the third approach directly, but we are doing it incrementally for motivation and explanation purposes. Since our implementation is in Java, we use Java syntax for describing operations defined by our approaches.

3.2 Brute Force Approach

This approach is to accommodate loggables that cannot provide any semantic information about a client request. The basic idea here is to record all the client requests in the log, and replay without modification to a new server. The main complication in this approach is the synchronization, which is described using *Figure 2*.

Initially, the logger and loggable start in the log and play modes, respectively. While in the play mode, the loggable treats each client request as just an encoded message that it knows nothing about, and simply relays the message as is to the logger. The loggable calls logMessage(Object message) to send the message to the logger. The logger logs each of these messages in chronological order. When the time comes to accommodate a latecomer, the loggable sends a REPLAY signal asking the logger to start replaying the logged messages. On sending/receiving the

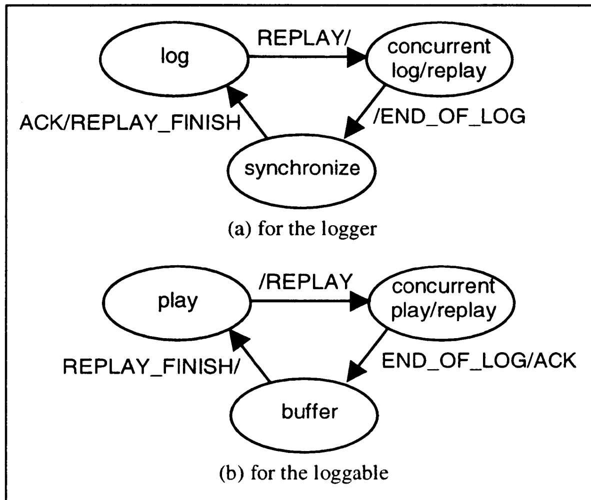


Figure 2. Mode Transition Diagram.

REPLAY signal, the loggable and logger go into the concurrent play/replay and concurrent log/replay modes respectively.

While in the concurrent log/replay mode, the logger simply replays the logged messages one by one to the loggable in exactly the same order in which they were logged. The loggable can, in turn, send these replayed messages directly to the latecomer's UI server.

While replay is going on, the client can send new messages to the loggable, which, in turn, sends them to the logger as well as to the pre-existing UI servers. The logger simply appends the new messages to the message log. When the logger has consumed the entire message log, it sends an END_OF_LOG signal to the loggable, and goes into the synchronize mode. The loggable responds to the END_OF_LOG

signal with an ACK signal, and goes into the buffer mode. The ACK signal defines a synchronization point, and it indicates to the logger that the loggable will send no new message until the logger replays all the messages sent so far and goes back into the log mode.

While in the synchronize mode, the logger waits for the ACK signal to arrive. While waiting, the logger can receive additional new client messages that the loggable may have sent before the ACK signal. The logger simply records these messages in the log, and replays them at the same time. When it finally receives the ACK signal, the logger sends a REPLAY_FINISH signal to the loggable, and goes back into the log mode. The buffer mode for the loggable is the same as the concurrent play/replay mode except that the loggable now buffers any new messages in its own temporary log. As soon as the loggable receives the REPLAY_FINISH signal from the logger, it sends the buffered new messages directly to the latecomer's UI server and to the logger, and then goes back into the play mode.

This brute force approach is inefficient because every client message has to be logged and replayed. This can consume a large amount of memory as the collaboration can last a long time. Additionally, the time needed to create the latecomer's user interface is strictly proportional to the entire collaboration time. If we have some semantic information about each of the client messages, we may be able to use less log space, and drastically reduce the time needed to create a latecomer's user interface. The following section describes a better approach assuming that the loggable can provide some semantic information about the client messages.

3.3 Independent Objects Approach

When we want to create the latecomer's user interface as quickly as possible, it is not necessary to show how the interface changed since the collaboration started. It is sufficient to show only the current state of the interface. For example, if the users changed the background color of a window many times, we do not have to save the history of all the background color changes, but only the latest background color.

Our approach here is based on observations such as this one, and its entire basic model of how the client makes changes to the UI server assumes that each of the client requests can be translated into one of the following abstract primitives.

- instantiate: The client instantiates some objects (e.g. windows, or active variables) on the UI server for interaction with the user.
- modify attributes: The client asks the server to change some attributes of an interaction object by specifying their new values. For example, the client can change the background color attribute of a window by specifying its new color value. An attribute value may not just be data, it may be a high level command that changes the attribute value. For example, it may be a command to draw a circle on a window foreground (which we assume to be modelled as an attribute of the window). The logger does not need to interpret the attribute values. It simply passes them to the loggable in the form it received them.
- destroy: When one of the interaction objects is no longer necessary, the client can send a request to destroy it.

For our approach to work, we require that the loggable translate the client's requests into the abstract primitives given above, and send them to the logger in order to report how the client is making changes to the user interface.

The logger maintains a record for each object instantiated by the client, and represents each attribute as a field of the record. Whenever the client modifies one of the attribute values, the logger replaces the old value with the new one for the corresponding field. The client can also modify attribute values of objects it did not instantiate when the objects are shared with other clients. For example, in the X window system, multiple clients can make changes to the shared default colormap. In this case, the logger depends on the loggable to report the objects as pre-existing before any modify primitives are applied to them. Since most of the client requests will involve changing certain attributes of interaction objects, this approach can accomplish drastic saving of the memory space needed to log the UI changes. Furthermore, we can accomplish more saving of log space by freeing memory allocated for objects that have been destroyed.

Now, when creating a latecomer's user interface, all the logger needs to do is just replay to the loggable the primitives to instantiate the objects (modify in case the objects are pre-existing) with the current attribute values, and the loggable can, in turn, translate the primitives into corresponding client requests and send them to the latecomer's UI server.

However, the model we assume in this approach so far may be too simple for some cases, since the client can make cumulative changes to an object attribute. For example, let us say that a logger and a loggable are working to provide latecomer service for a text editor application. The loggable has modelled the text window as a single object on the logger, and the foreground of the text window as a single attribute of the object. The loggable's intention is to translate a client request that displays characters on the window into a modify attribute - the loggable uses the request itself (without modification) as the value parameter of a modify primitive that changes the foreground attribute. When the "value" is replayed later, the loggable can simply send the "value" to the latecomer's UI server. The text editor client can display characters one after another through different text drawing requests as the users enter them using the keyboard. Instead of refreshing the whole window foreground each time a new character needs to be displayed, each request can change a part of the screen where the new character needs to be displayed. We refer to such a value change as cumulative since it cannot replace the previous value of the associated attribute. We extend our model to allow the loggable to specify the cumulative nature of some attribute value changes.

With this change in our model, the logger now looks at the cumulative nature of the value change, and if it is cumulative, it simply appends the new value to the previous value of the associated attribute. When a non-cumulative (replacing) value change needs to be applied to an attribute, the whole list of values attached to the attribute is replaced by the new value. During replay, the logger replays the value changes in the list one by one in chronological order.

Another good use of this extension arises when the loggable wants the logger to log the history of value changes even when they are not actually cumulative in

nature. For example, as mentioned before, a latecomer may want to view the history of all the changes in a certain window.

Hence, the log space we need to use is more or less proportional to the number of objects that the client is maintaining on the UI server, but not exactly proportional due to some cumulative value changes. Moreover, the time needed to create the latecomer's user interface is not necessarily subject to the length of the total collaboration time.

The mode changes that the logger and the loggable take are basically the same as in the brute force approach. However, the logger in the concurrent log/replay mode works a little differently in how it deals with new primitives coming from the loggable.

If the new primitive involves an object that has already been set up on the latecomer's UI server, the new primitive is simply applied to the object record for a future latecomer accommodation service, and also sent to the loggable as a replay primitive. If the new primitive is about an object that is yet to be set up on the latecomer's UI server, it is simply applied to the object record, but is not sent to the loggable as a replay primitive. This is because the new primitive's effect will be replayed later when the logger gets to the associated object. Finally, if the primitive is about an object which the logger is in the middle of setting up on the latecomer's UI server, the application of the primitive to the record is temporarily delayed until the object is completely set up, and when the set up is complete for the object, the delayed primitives are applied to the object record for a future latecomer accommodation service, and sent to the loggable one by one in the order that they were received by the logger.

Our independent objects approach subsumes the previous brute force approach. If the loggable is sending client messages using the model assumed in the brute force approach, we simply instantiate one log object with one attribute, and we treat each client message as a cumulative value change to the attribute of the log object, thereby simulating the brute force logging of the previous approach.

3.4 Dependency Handling Approach

We saw in our previous approach that cumulativeness of attribute value changes enforces an order during replay. A cumulative value change for an object attribute cannot be replayed unless its previous value change is first replayed. Put another way, cumulativeness of an attribute value change forms a replay dependency of the attribute value change on its predecessor. Since this particular form of dependency is made within the same object, we call it an intra-object dependency (or cumulative dependency). In this subsection, we extend our model to include clients issuing primitives that create other kinds of dependencies (i.e. inter-object dependencies), and describe how it affects our log and replay approach. We first motivate the need to make such dependencies.

Windows for some applications often have repetitive tiles of an arbitrary pattern as the background. If the application client were to draw the repetitive pattern on the entire window, it would be very cumbersome and error-prone. To make programming easier in such a case, window systems such as X provide the programmers with a scratch-pad-like abstraction, called a pixmap, which itself is not

directly displayable, but can be used in combination with other abstractions such as windows. To create a window with a tiled background in X, the X client would first draw the unit pattern on a pixmap, and modify the window's background attribute to refer to the pixmap. In response to the request, the X server can copy the pixmap content and repeatedly draw the pattern on the window. Later, the client can modify the pixmap to use it for some other purposes, or delete it when it is no longer needed without affecting the background pattern of the window.

In this example, the tiled window (depender) object is said to depend on the pixmap (dependee) object. Such dependency relationships between different objects (inter-object dependencies) affect how we log and replay the client request primitives in the following ways.

First, dependencies affect the order in which we instantiate and modify objects on the latecomer's UI server during replay. In our example, primitives that create the pixmap, and update it with the attribute values should be replayed before the primitive that forms the window's dependency on the pixmap.

Second, a destroy primitive from the loggable does not mean an immediate removal of the associated object record from the logger, because some other object may have formed a dependency on it. If we do remove the record, we would not be able to build the dependee object on the latecomer's UI server at all, which, in turn, means that the depender object cannot have one of its attributes set correctly.

We represent each of the dependencies as a directed edge that has its tail on the depender object node and its head on the dependee object node. The graph thus created can be used to replay client request primitives in the correct order: i.e. replay the dependee object first, and then the depender object. Also, when we delete a node in response to a destroy primitive, we check whether an edge has its head on the associated node: i.e. whether any other object depends on it. If there is indeed a depender object, we do not delete the object node, but we just mark it as destroyed. If there is no depender object, we delete the object node along with all the edges that originate from the deleted node. Then, we recursively follow the deleted edges to find and delete any node that could not be deleted because its depender node still existed.

There is another, related, problem to be resolved, however. This problem occurs when the client tries to modify a dependee object. We cannot apply the attribute value changes implied in the modify primitive to the object because we assume that its depender object depends on all the current attribute values of the object. We resolve this problem by object versioning. We consider the instantiation of an object as creating the first version of the object, which becomes its current version. Later, the loggable may send a primitive that modifies the object. In response to this primitive, the logger first finds out whether any other object depends on the current object to be modified. If there is no such object, the logger applies the change to the current version of the object. Otherwise, the logger creates a new version copy of the current version. The new version is a copy of the previous version except that all its attribute values are initialized to null. The logger then applies the change to the new version, which becomes its current version. To indicate the replay order of the two versions, we create a dependency of the new version on the previous version. We refer to this dependency as version dependency. Any subsequent modification made

to the object is about the most current version of the object. With the introduction of versioning, the dependency graph is now made up of different versions of objects, some of which are connected by version and inter-object dependencies. Version dependency imposes another restriction on how we replay primitives: an object version cannot be replayed unless its previous version and all other object versions that depend on the previous version have been replayed. This is because we lose the previous version's context by replaying its next version.

Versioning not only preserves the dependee object context for the depender object, but also removes cycles in the dependency diagram. Without versioning, cycles can occur. For example, when creating a pixmap in X, the X request should specify a hierarchy of windows for which the pixmap can be used, by including, as an attribute of the pixmap, a window that belongs to the window hierarchy. We refer to this window as the reference window of the pixmap. So let us say that when creating a pixmap A, the client specified a window B, thereby creating a dependency of pixmap A on window B. Now, after the client draws some basic pattern on pixmap A, it designates pixmap A as the background pattern of window B. Assuming there is no versioning, this modification creates a dependency of window B on pixmap A. Thus, we have a cycle formed with the two object nodes and the

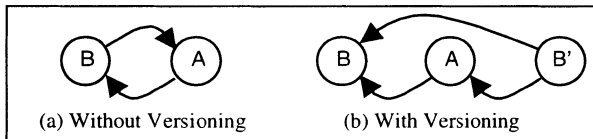


Figure 3. Cycle Removal through Versioning.

two dependency edges between them. Such a cycle creates a problem when we replay primitives to create objects on a latecomer's UI server: we cannot create window B without creating pixmap A, or vice versa, because they depend on each other. Deleting nodes creates a similar problem. Let us say that a primitive to destroy the window arrives. It cannot be deleted because pixmap A depends on it. Now, if a primitive to destroy pixmap A arrives, it too cannot be honored, since the undeleted window B depends on it.

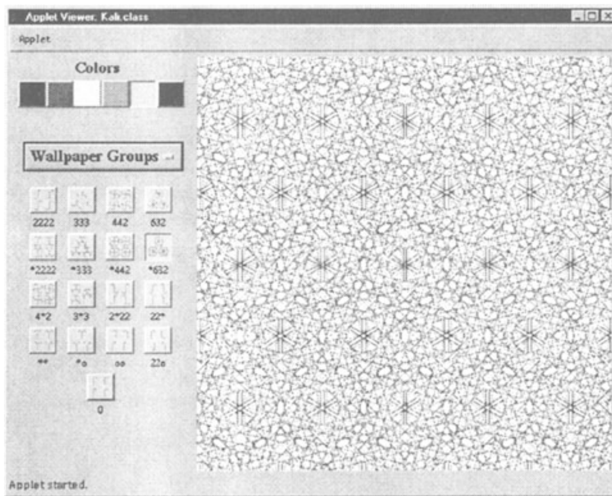
Let us illustrate how versioning works to prevent such a cycle. When making the background pattern attribute change in window B, we find that pixmap A depends on it. Instead of applying the new value in the current version of window B, we create a new version B' of window B and apply the value change to B'. Since we are effectively adding a new node (which does not have any incoming edge) and making the dependencies from there, we never have a cycle in the dependency diagram. The versioning process is illustrated in *Figure 3*.

With the guarantee that there is no cycle in the entire object dependency diagram, we can use the same procedures described above for logging and replaying.

Our dependency handling approach follows basically the same mode changes as described before. However, the logger in the concurrent log/replay mode works a little differently in how it deals with each of the new primitives coming from the loggable. Unlike the previous approach, which applies the new primitive's value

changes to the logger data structures and optionally sends the primitive on the fly depending on whether the associated object has been replayed, we take a rather simple solution for now. We log any new primitive coming from the loggable in a temporary log. When we finish replaying all the versions in the dependency diagram, we take each primitive in the temporary log, and send it back to the loggable. Before sending the primitive, we also apply the primitive's value changes to the dependency diagram in order to prepare for a future latecomer accommodation service. We take this simple approach because the new primitive's effect during replay is not necessarily confined to a single object version due to inter-object and version dependencies, and it is probably not worth the effort to try to determine the new primitive's effect during replay when we can simply apply it to the data structures and send it back to the loggable when the replay is over.

Our new dependency handling approach subsumes our previous independent objects approach in a straightforward manner, since the previous approach is just a special case of our new approach, where there is no dependency among different objects.



3.5 Evaluation and Implementation

Figure 4. A User Interface Snapshot of an Application Program Kali

Let us evaluate our latecomer accommodation service framework based on the requirements described in the introduction. It is composable by design as shown in its architecture (*Figure 1*), and it imposes no policies or protocols that are unrelated to latecomer accommodation support. It compresses the log by design, though the extent of the compression depends on the amount of semantic knowledge provided

by the loggable. It is easy to program since the loggable takes on a translation role instead of implementing the entire service. However, there is a trade-off between log compression and ease of programming - the more compression the service provides, the more work the loggable has to do in order to provide more semantic information. But, we still believe that even with the dependency handling approach, the effort required to build a loggable will be less than what is required to implement the log compression. For example, it is easier to describe the dependencies than to interpret them. Indeed, compression comes at a cost, but the loggable only has to specify the parameters of the compression algorithm. The framework is generic because it does not make any assumptions about the UI abstraction. For instance, the abstraction could be a Suite active variable, or an X window system window.

We have implemented the logger in Java. So far in our paper, only X examples were used to motivate the description of our work. To verify that our logger implementation is useful for higher-level abstraction systems beyond X, we added latecomer accommodation service to a Java application called Kali.

Figure 4 shows a snapshot of the user interface created by the application program Kali. The left portion of the interface is the control panel, while the right portion is the canvas on which a user can draw the pattern that he desires. The control panel is divided into the color panel and the group panel. The color panel controls what color is used to draw on the canvas while the group panel is used to

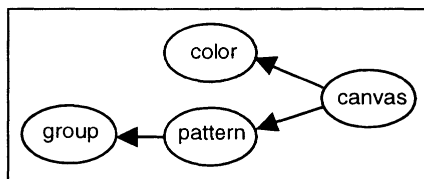


Figure 5. Dependency Relationship among Kali State Objects.

set a pattern to lay on the canvas. There are three alternative groups of patterns a user can choose from a drop-down group menu, and for each different group selection, the application displays a different set of patterns that the user can choose as buttons below the group menu. Selection of a color or a pattern affects only subsequent drawing the user lays on the canvas, but does not influence previous work. No effect is seen when the group is changed, but the canvas is cleared when a particular pattern is chosen for the first time after the group change.

As illustrated in *Figure 5*, there are basically four states maintained within the application program: i.e. those of color, group, pattern, and canvas. The loggable models these states as four state objects, each of which has a single attribute used to log events that change the associated state object. Each selection in the color buttons, the group menu, and the pattern buttons is modeled as a single event, and it is non-cumulative since its effect on the associated state object replaces that of a previous selection event. Each event associated with the canvas state object is cumulative since its effect adds to the current drawing of the canvas, except that the first canvas event after each new group selection followed by a pattern selection is

non-cumulative. *Figure 5* also illustrates the inter-object dependencies the loggable specifies to enforce a correct replay order.

As can be seen from the nature of the application, logging is a more attractive alternative since state capture is rather complex. The state is not only dispersed in the canvas but also in the control panel since the effect of the next event in the canvas depends on the state in the control panel. The importance of this method is that by clearly defining the dependencies which are natural to the application, and the properties of the attributes being logged, the entire compression is automated. Moreover, the change to the application program was minimal – only a small portion of event handling code within the program had to be changed to both send events to the logger and to receive replayed events.

We ran the application using the logger for latecomer accommodation and obtained the following statistics. *Table 1* covers the entire spectrum where we use all the events that are logged while in other cases, only necessary events are replayed.

In the first case, the control panel was used to change only the color. In the second case, the group of patterns in the control panel was changed. In the last case, the entire canvas was cleared and a design recreated. We had two Color events before the group was changed to redraw a new design. In the replay, all the events that occurred before the pattern change, which include the two color events, are purged and we can see effective log compression. The number of replayed events are exactly equal to the events that occur after the last clear occurred and hence offer an advantage of a compact log.

Table 1. Logger Performance on Log Size Reduction

Logged Events					Replayed Events		
Canvas	Button	Group	Color	Total	Canvas	Control Panel	Total
387	0	0	3	390	387	3	390
229	0	1	3	233	229	4	233
463	1	1	4	469	215	4	219

4. FUTURE WORK

In both of our latter two approaches, the loggable can have the logger replay the whole history of a certain window's state changes by specifying all the drawing requests to the window as having the cumulative characteristic. However, just replaying the value changes of an object's attributes (e.g. just showing what went on within only the main window of a drawing application) may not be sufficient for some latecomers. A latecomer who is also a novice user of the shared client may want to learn how to work with the client's interface while watching the request replay, such as which menus are used. But simply replaying primitives according to the dependency relationships cannot satisfy such needs, because some of the windows used for menus may be temporarily created, used by the user, and destroyed immediately, and hence immediately removed from the data structure. We

plan to define a playback dependency, which basically defines a playback sequence of window drawing requests.

We also plan to use our system to provide a composable latecomer service for a variety of systems. Since our implementation is coded in Java, it is easiest to compose it with systems implemented in Java. We plan to compose it with the Java applications we plan to build as part of the Collaboration Bus project going on in our department.

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation Grants IRI-9408708, IRI-9508514, IRI-9627619, and CDA-9624662, and by DARPA/ONR grant N 66001-96-C-8507.

REFERENCES

Abdel-Wahab, H. M. and Feit, M. A., XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration, *Proceedings, IEEE Conference on Communications Software: Communications for Distributed Applications & Systems*, Chapel Hill, NC, pp. 159-167, April 1991.

Chung, G., Jeffay, K., and Abdel-Wahab, H., Dynamic Participation in Computer-based Conferencing System, *Journal of Computer Communications*, 17(1): 7-16, January 1994.

Dewan, P. and Choudhary, R., A high-level flexible framework for implementing multi-user user interfaces, *ACM Transactions on Information Systems 10*, 4, 345-380, October 1992.

Manohar, N. R. and Prakash, A., The Session Capture and Replay Paradigm for Asynchronous Collaboration, *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, September 10-14, Stockholm, Sweden, 149-164, 1995.

NCSA, Habanero, http://www.ncsa.uiuc.edu/SDG/Software/Habanero/.

Roseman, M. and Greenberg, S., Building Real Time Groupware with GroupKit, A Groupware Toolkit, *ACM Transactions on Computer Human Interaction*, 1996.

BIOGRAPHY

Goopeel Chung is a Ph.D. student in the Department of Computer Science at the University of North Carolina at Chapel Hill. He received a B.S. degree in Computer Engineering from Seoul National University, and a M.S. degree in Computer Science from University of North Carolina at Chapel Hill. His research interests are flexible shared window systems and process migration.

Prasun Dewan is a professor in the Department of Computer Science at the University of North Carolina at Chapel Hill. Before joining UNC-Chapel Hill, he was on the faculty of Purdue University. He received a B.Tech. degree in Electrical Engineering from the Indian Institute of Technology of New Delhi and a Ph.D. in

Computer Science from University of Wisconsin at Madison. His research interests are in frameworks for implementing single-user and multi-user applications, collaborative software engineering, object-oriented databases, operating systems, process migration, mobile computing, and interoperability.

Sadagopan Rajaram received the B.Tech degree in Computer Science and Engineering from the Indian Institute of Technology, Madras (Chennai), in 1997. He is currently working towards his Masters degree in Computer Science from the University of North Carolina at Chapel Hill. His interests include CSCW, particularly in the area of access control and security.

Discussion

Ken Fishkin: I have a question about your loggable filtering. If one views an event as a command to a virtual machine, then the event log is like a sequence of commands in assembly language. So, the filtering you describe is similar to classic code optimisation. For instance, some of the techniques you describe in the paper are similar to peephole optimisation, dead code removal, and code hoisting. Would it be useful to strengthen this similarity and investigate the use of available code optimisation packages?

Prasun Dewan: Our system is flexible, and whoever implements the loggable filter certainly can do any optimisation. I need to think about this similarity, however.

Christian Gram: Can you log "Undo" and manage it without losing information?

Prasun Dewan: Yes, I think it should work.

John Grundy: Is Kali a single-user application?

Prasun Dewan: Yes. We is turned it into a multi-user application, then added latecomer management.

Nick Graham: You seem to be biased towards applications that are WYSIWIS, and artifact-based, where artifacts can be represented by a set of graphical calls.

Prasun Dewan: As to WYSIWIS, yes. As to artifacts, I do not quite agree. It is up to you to describe how your application is to be rendered.

Len Bass: How would you handle users with different levels of synchronisation? Would you have one logger for each?

Prasun Dewan: Yes, we would have one logger per user in that case.

Stephane Chatty: Can you do fast-forward and rewind through your event logs?

Prasun Dewan: We can log enough events to support these operations, but have not implemented them so far.

Franck Tarpin-Bernard: After a long session , what is the size of a log file? Isn't it very large?

Prasun Dewan: If you want fast-forward and rewind, then yes it would be very large, since you would have to log all events. Otherwise, you can have checkpoints and shorten your file.