Consistency Control for Distributed Interactive Media

Jürgen Vogel, Martin Mauve
Praktische Informatik IV, University of Mannheim, Germany
{vogel,mauve}@informatik.uni-mannheim.de

Abstract— **In this paper we present a generic consistency control service for distributed interactive media, i.e. media which allow a distributed group of users to interact with the medium itself. Consistency control is vital to these media since they typically require that a local copy of the medium's state be maintained by each user's application. Our service helps the applications to keep the local state copies consistent. The main characteristics of this service are as follows: a significant number of inconsistencies are prevented by using a mechanism called local lag. Inconsistencies that cannot be prevented are repaired by an improved timewarp algorithm that can be executed locally without burdening the network or the applications of other users. Exceptional situations and consistency during late-join situations are supported by a consistent state request mechanism. Moreover, the service also supports the application in detecting intention conflicts between the actions of distinct users. The major part of this functionality is based on a media model and the application level protocol for distributed interactive media (RTP/I) and can thus be reused by arbitrary RTP/I-based applications. In order to demonstrate the feasibility of our approach and to evaluate its performance we have integrated the generic consistency service into a shared whiteboard system.**

Index Terms—**Consistency, Distributed Interactive Media, RTP/I, Timewarp, Local Lag, Intention Conflict, Late Join, mlb.**

I. Introduction

Distributed interactive media are media which allow a set of spatially separated users to interact synchronously with the medium itself. Typical examples of distributed interactive media are shared whiteboards, which are used to present and edit slides in a teleconferencing environment [Tun98], [GE98], distributed virtual environments (DVEs) [Hag96], shared text editors [HC97], and computer games with network support [GD98].

In order to provide high responsiveness and to avoid the drawbacks of centralized approaches, such as the presence of a single-point-of-failure and lack of scalability, applications for distributed interactive media often employ a replicated distribution architecture.

In this architecture each user runs an instance of the application, which manages a local copy of the medium's shared state. For example, the state of a shared whiteboard presentation includes a number of presentation slides, each containing graphical objects such as images and text. User actions (e.g., moving an image or inserting new characters into a text object) can change this state. Local user actions therefore have to be transmitted to all remote instances of the application so that these can modify their local copy of the state accordingly.

Without taking special precautions, the consistency of the shared state cannot be guaranteed, even if all user actions are successfully delivered to all instances of the application. The main problem is that the transmission of an action is subject to a certain network delay. While a user action can be executed at once at the originating site, it takes some time to transmit it over a network to the other instances. Therefore, execution of the operation is delayed at the remote instances. This can result in different orderings of operations, thereby inducing an inconsistency. Consider, for instance, two participants of a whiteboard session each changing the color of a rectangle almost at the same time. The first participant changes the color to red, the second to blue. Given a significant network delay, it is likely that once both actions have been executed, the rectangle shown by the application of the first participant is blue, while the rectangle presented by the application of the second participant is red.

Generally, existing mechanisms to prevent these problems can be classified as *pessimistic* or *optimistic* approaches. Pessimistic mechanisms prevent inconsistencies, typically by using locking algorithms or floor control to avoid simultaneous conflicting operations [MD96]. Even though this technique is very effective, pessimistic approaches have one major drawback: real collaboration between session participants is restricted. Optimistic approaches, on the other hand, allow inconsistencies to happen and seek to repair them afterwards in an efficient way [EG89], [SJZ$^+$98]. While these mechanisms support collaboration between users, existing approaches tend to be very complex, and application-dependent, and to expose the user to frequent short-term inconsistencies.

In this paper we present an optimistic consistency control service that improves several important aspects of existing approaches:

- It eliminates a significant number of short-term inconsistencies by using a method called local-lag.
- It employs an improved timewarp algorithm to repair inconsistencies. This algorithm can be performed locally without additional burden on the network, while minimizing the local computational effort for executing the repair of inconsistencies.
- It takes into account participants that join an ongoing session, i.e. it supports late joiners.
- It provides mechanisms to detect conflicts of intention between user actions.
- It has been designed and implemented as a generic service reusable for arbitrary distributed interactive media which employ the Real-Time Protocol for Distributed Interactive Media (RTP/I) [MHKE01].

To demonstrate the feasibility of our ideas and to evaluate the performance of the consistency service, we have integrated it into a shared whiteboard, the multimedia lecture board (mlb) [Vog01a].

The remainder of this paper is structured as follows. In Section Two we introduce a model for distributed interactive media which allows an application-independent discussion of common problems and solutions. Furthermore, we give a brief overview of the RTP/I protocol. In Section Three consistency criteria for distributed interactive media are defined. The subsequent Sections present the concepts of local lag, timewarp, and state request, and show how we combine these mechanisms to form a generic consistency control service. For each element we explain how the service cooperates with the mlb. In Section Seven we investigate how the service is able to detect conflicting user intentions. In Section Eight we present a performance evaluation of the consistency service and its application to the mlb. Related work is examined in Section Nine. The paper concludes with a summary and an outlook on future work.

II. Media Model and RTP/I

In order to provide a generic service that is reusable for a whole class of media, we give a brief overview of the characteristics of the distributed interactive media class. A more detailed discussion can be found in [MHKE01].

A. States and Events

A distributed interactive medium has a state. For example, the state of a shared whiteboard is defined by the content of all pages present in the whiteboard. In order to perceive the state of a distributed interactive medium, a user needs an application, e.g., a shared whiteboard application is necessary to see the pages of a shared whiteboard presentation. This application generally maintains a local copy of (parts of) the medium's state. Applications for distributed interactive media are therefore said to have a replicated distribution architecture. For all application instances the local state of the medium should be at least reasonably similar. It is therefore necessary to synchronize the local copies of the medium's state among all participants, so that the overall state is kept consistent.

The state of a distributed interactive medium can change for two reasons, either by passage of time or by events. The state of the medium between two successive events is fully deterministic and depends only on the passage of time. Generally, a state change caused by the passage of time does not require the exchange of information between application instances, since each user's instance can independently calculate the required state updates. An example of a state change caused by the passage of time is the animation of an object moving across the screen.

Any state change that is not a fully deterministic function of time is caused by an event. Generally, events are (user) interactions with the medium, e.g., the user makes an annotation on a shared whiteboard page. Typically, information about events needs to be transmitted to all remote instances of an application in order to keep all state copies up to date. In the following, we use the term operation to identify states or events that are transmitted to inform remote instances of the application about the actions of a local user. Applications that allow their state to change only because of events are called *discrete* (e.g., shared whiteboards), while applications supporting changes by both events and the passage of time are called *continuous* (e.g., distributed virtual environments).

B. Partitioning the Medium - Sub-Components

In order to provide for a flexible and scalable handling of state information, it is desirable to partition an interactive medium into several sub-components. In addition to breaking down the complete state of an interactive medium into more manageable parts, such partitioning allows the participants of a session to track only the states of those sub-components in which they are actually interested. Examples of sub-components are 3D objects (an avatar, a car) in a distributed virtual environment, or the pages of a shared whiteboard presentation. Events affect only the state of their target sub-component(s).

C. RTP/I

The Real Time Protocol for Distributed Interactive Media (RTP/I) [MHKE01] is based on the media model described above. While our implementation of the generic consistency service makes use of RTP/I, the concepts presented here may also be used by applications which use other application-level protocols.

RTP/I is a protocol framework for distributed interactive media. It consists of two parts: a data transfer protocol for the transport of events, states, and requests for state information in the form of so-called application data units (ADUs), and a control protocol for meta-information about the medium and the participants of a session.

The data transfer protocol (RTP/I) provides a standardized framing for ADUs. This framing contains information that is common to the distributed interactive media class. It can be used by a generic service to interpret the semantics of an ADU without needing to know its medium-specific encoding. Typical examples of the information contained in the RTP/I data framing are a timestamp that indicates at what time an event happened or a state was calculated, an identifier for the affected sub-component, the type of the data (e.g., event or state), and the unique ID of the ADU's sender. RTP/I is also used to request the state of a sub-component in a standardized way.

The RTP/I control protocol (RTCP/I) conveys information about the participants of a session, e.g., the participants' names and email addresses. This information can be used to establish a light-weight session control. Moreover, RTCP/I provides information about the sub-components that are present in a session. Information about each sub-component is announced regularly.

RTP/I is not a complete protocol. It needs to be adapted to the requirements of a specific medium by means of a payload type definition. Essentially a payload type definition describes how the medium specific data are encoded. A payload type definition for shared whiteboards as it is used by the multimedia lecture board can be found in [Vog01b].

RTP/I is closely related to the Real Time Transport Protocol (RTP) [SCFJ99], which is mainly used for the transmission of audio and video. However, while RTP/I reuses many aspects of RTP, it has been thoroughly adapted to meet the needs of distributed interactive media.

D. RTP/I and Reliability

RTP/I itself does not specify any reliable transport mechanisms, even though such mechanisms are required by many distributed interactive media [MH00]. Instead reliability is orthogonal to RTP/I, enabling the use of application-level reliability or transparent reliability as preferred by the application. The multimedia lecture board relies on the latter approach by using the scalable multicast protocol (smp) [GE98], which provides both reliability and source ordering. In the following, we expect that operations are eventually delivered to each application instance through whatever method is preferred by the application.

III. Consistency in Distributed Interactive Media

Consistency in distributed interactive media is about finding an order in the sequence of operations and ensuring that the state of the medium in all application instances looks as if all operations had been executed in that order. Furthermore, if the medium is continuous, it is also important to take into account the point in time at which an operation should be executed.

In order to give a consistency criterion we define a total ordering relation based on physical clocks. We assume that the physical clocks of all application instances are reasonably synchronized [1] (using e.g., NTP [Mil92] or GPS clocks).

[1]A complete synchronization of all physical clocks is not necessary since differing clocks do not endanger the consistency criterion defined below. However, unsynchronized clocks increase the probability of short-term inconsistencies and reduce the fairness among participants.

Definition 1 (Partial physical-time-ordering relation $<$) Given two operations O_a and O_b with timestamps T_a and T_b, then O_a is said to happen before O_b, expressed as $O_a < O_b$, iff $T_a < T_b$.

Definition 2 (Simultaneous operations $\doteq$) Any two operations O_a and O_b with timestamps T_a and T_b are said to be simultaneous, expressed as $O_a \doteq O_b$, iff $T_a = T_b$.

The partial physical-time-ordering relation can be extended to become a total ordering relation by using a tie-breaker for simultaneous operations. Examples of tie-breakers are IP addresses for simultaneous operations from different participants, and a counter for operations originating from the same participant.

Definition 3 (Total physical-time-ordering relation $\ll$) Given two operations O_a and O_b with timestamps T_a and T_b and tie-breakers B_a and B_b, then $O_a \ll O_b$, iff (1) $T_a < T_b$ or (2) $T_a = T_b$ and $B_a < B_b$.

Based on the total physical-time-ordering relation the consistency criterion for distributed interactive media can be defined as follows:

Consistency Criterion A distributed interactive medium is consistent, if after all operations have been executed at all sites, the state of the medium at all sites is identical to the state which would have been reached by executing all operations in the order given by the complete physical-time-ordering at the physical time denoted by the timestamps of the operations.

For continuous interactive media it is crucial to consider the physical time of an operation, since state changes can happen because of the passage of time. Therefore, an event may have distinct effects depending on the point in time at which it is executed. However, it is important to note that the consistency criterion does not necessarily require operations to be actually executed at the correct point in time. It is perfectly legal to use a repair mechanism which calculates the correct state as if the operations had been executed at the correct physical time. Furthermore, it can be easily seen that in the discrete domain the consistency criterion is reduced in order to specify an ordering of operations.

If a mechanism is used which ensures the consistency criterion for distributed interactive media, the *consistency* of a medium is guaranteed. However, the resulting state of the medium may violate the original *intention* of the participants. For example, consider two participants of a whiteboard session each changing the color of an object almost simultaneously. After execution of both events, the object will be in a consistent state, meaning that each participant will see the same color. This is true since we assume that there exists a mechanism which ensures the consistency criterion. Nevertheless, the intention of one participant will be violated. The consistency service presented in this paper detects intention conflicts and allows the application to inform the users about their occurrence.

A more detailed discussion of consistency criteria can be found in [Mau00], [SJZ$^+$98] and [EG89].

In the following sections we describe our generic consistency service, which uses a combination of algorithms to ensure the consistency criterion. First, *local lag* is used to reduce the number of inconsistencies. Second, *timewarp* repairs inconsistencies exceeding the time-span covered by local lag. Third, *state request* repairs inconsistencies exceeding the time-span covered by timewarp.

IV. Local Lag

Inconsistencies in distributed interactive media are caused by the varying period of time, called *operation delay*, between the time an operation is issued by the user and the time that operation is executed (see Figure 1 (a)). While at the originating application instance there is virtually no operation delay, the remote instances will experience an operation delay caused by the time it takes for the operation to reach the remote instance over the network. This time may be significantly extended if the network drops packets and these need to be retransmitted. If the distributed instances of an application experience different amounts of operation delay, the shared state is in danger of becoming inconsistent. Even if we assume that these inconsistencies will eventually be repaired by mechanisms ensuring the consistency criterion they have a negative impact: First, the user perceives an inconsistent state. Second, the application has to calculate the correct state, which might consume significant computational resources. Finally, when the application displays the corrected state it might be considerably different from the (wrong) state visible before, causing artifacts such as jumping objects.

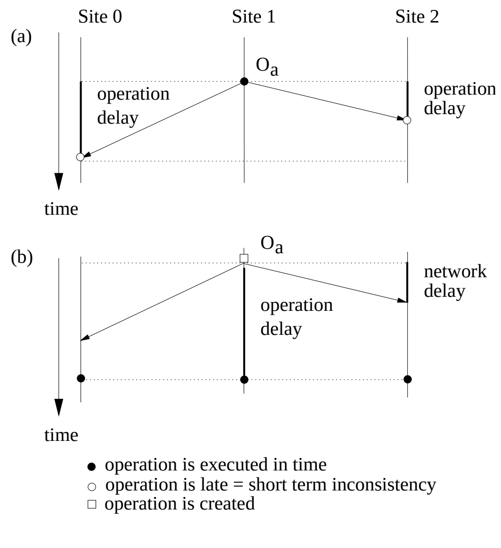

Fig. 1. Regular and equalized operation delay

Therefore it is desirable to prevent such *short-term inconsistencies* from happening. One possible way to do this is to equalize the operation delay for all instances of the application. The equalization is done by shifting the execution timestamp T_a of an event O_a into the future, i.e., by introducing an artificial delay. The delay period is used to distribute the operation to all application instances. In the optimal case, distribution will have been completed before T_a is reached, allowing all instances to execute O_a at the correct time (see Figure 1 (b)). Because of the artificial delay introduced by the originating site, we call this approach *local lag* [Mau00], it is related to the bucket synchronization mechanism employed by MiMaze [GD98].

Choosing the right value for the local lag is not an easy task. Enlarging its value will increase the probability that short-term inconsistencies can be repaired (because of the longer reordering time-span), but will decrease the *responsiveness* of the medium, since users have to wait longer until they see the effect of their actions. A compromise in this trade-off situation has to take into account the expected average network delay and the maximum tolerable response time for user actions. Depending on the operation, the latter seems to lie between 50 ms and 300 ms [Mau00], [VGB99], which is sufficient for continental or even worldwide sessions, with expected network delays of 40 ms and 100 ms respectively.

Implementation of the local lag concept as a generic service is straightforward. The application hands over all local and remote operations in the form of RTP/I application data units (ADUs) to the generic service, where they are inserted into a local lag queue. Since the ADUs contain all the necessary information, the queue can be sorted by the total physical-time-ordering relation as defined in Section 3. If the execution timestamp T_a of an operation O_a is reached, the generic service tells the application to execute O_a.

It is interesting to note that the use of the local lag concept leads to a new programming paradigm. Traditionally, the functionality triggered by a local event is as follows: execute the event and display the new state, then create and distribute the corresponding ADU. With local lag this changes to: create and distribute the ADU, insert the ADU together with received ADUs into the local lag queue, wait until execution time is reached, then calculate and display the new state. Therefore, with local lag, remote and local operations need no longer be distinguished once the ADU has been created and enqueued.

Even though local lag can reduce the number of short-term inconsistencies significantly, inconsistencies can still occur. Indicative of a possible inconsistency is the receipt of an operation O_a with $T_a > T_C$ where T_C is the current time. In the next section we present timewarp as an efficient repair mechanism for those inconsistencies that cannot be not prevented by local lag.

V. Timewarp

With timewarp [Edw97], [Mau00] an application instance can repair inconsistencies on the basis of information stored locally. The main benefit of this approach then is that it neither increases the total network load nor burdens the other session members.

A. Basic Timewarp

The basic idea of timewarp is that each application instance save the state of the distributed interactive medium periodically. Moreover, all operations (local and remote) up to a certain point in time are stored as well, thus building a *history*. If an inconsistency occurs, the medium is rolled back to the last state saved before the operation should have been executed. Then the operation which caused the inconsistency is inserted into the history. After that the medium is played in fast-forward mode, executing the operations from the history at appropriate times until T_C is reached and operation is resumed at normal pace. To avoid confusion, only the repaired state of the medium should be visible to the user.

The main drawback to this approach is that it requires a significant amount of computational complexity to perform the fast-forward calculation of the repaired state. Furthermore, saving the state of a medium at regular intervals might consume a more than negligible amount of memory. Finally, the application has to support the timewarp, which might increase its complexity. In the following we present an improved timewarp algorithm which:

- minimizes the number of timewarps in order to reduce the computational burden,
- reduces the amount of memory required by switching to a different repair strategy if the offending operation is received exceedingly late, and
- supports the application by providing a large part of the required functionality as part of the generic consistency service.

B. Improved Timewarp

Since a timewarp can be costly in terms of application performance, it is desirable to improve the basic algorithm, so that a timewarp is executed only when absolutely necessary. This can be done by making use of both RTP/I-level and application-level knowledge.

The partitioning of the medium can be utilized to limit the range of a timewarp. Instead of recalculating the complete state, only the sub-component affected by the late-arriving operation has to be time-warped. For example, the manipulation of a graphical object on a whiteboard page concerns only that object and, at most, all other objects on that page (and the page itself), but leaves the rest of the whiteboard's state untouched. Taking the partitioning of the medium into account improves the timewarp in two respects. First, it limits the amount of state information that has to be recalculated during a timewarp. Second, as we shall see later on, it reduces the operations to be taken into account to decide whether or not a timewarp is required.

In the discrete domain, there exist quite a few cases where a timewarp is not necessary and the late-arriving operation O_a can either be ignored or executed immediately without a timewarp (see Figure 2). For the discrete domain the improved timewarp first checks if there exist operations O_i with timestamp T_i so that $T_a \ll T_i < T_C$. If not, O_a can be executed without endangering consistency. Second, in case there is a (non empty) set of operations with $T_a \ll T_i < T_C$, we examine for each O_i if it conflicts with O_a. In this context, two operations are defined as conflicting if they change the same aspect of a sub-component.

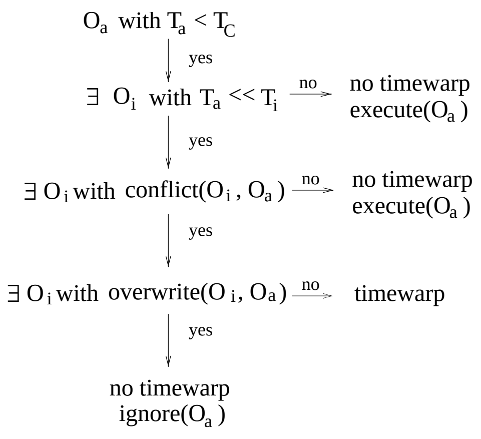

Fig. 2. Decision algorithm for improved timewarp

For example, events changing the object on a whiteboard page conflict only if they concern the same object and the same attribute of that object (e.g., color, size, position). In order to decide whether two operations for the same sub-component are in conflict, the application has to provide an appropriate function `conflict(`O_i`, `O_a`)` which is called by the generic consistency service. Figure 3 outlines this function for the mlb. In essence, the conflict function for the mlb decides that two operations are conflicting if: (1) they are executed on the same object and change aspects of the object which are identical or dependent on each other, (2) the two objects share the same parent (a parent is a grouping of objects) and the stacking order or visibility information of the objects is changed, or (3) both objects change their parent to the same new parent. It is important to realize that any application may start using the generic service with a very simple conflict function, which returns `true` in almost all situations. Later on this function may then be improved to prevent more timewarps. For the mlb we have noticed that a relatively simple conflict function already reduces the number of timewarps siginificantly.

```
conflict($O_i$, $O_a$):
1. objectID($O_i$) = objectID($O_a$)
   – $O_i$ is state or $O_a$ is state
   – $O_i$ is delete event
   – $O_i$ and $O_a$ are change events
      · subType($O_i$) = subType($O_a$)
      · objType ∈ {polygon, polyline}
        and subTypes ∈ {move point, add point}
      · objType = text and subTypes ∈ {change font,
        insert char, delete char, change cursor}
2. objectID($O_i$) ≠ objectID($O_a$)
   and parent($O_i$) = parent($O_a$)
   – $O_i$ and $O_a$ are state and overlap($O_i$, $O_a$)
   – $O_{a/i}$ is state and $O_{i/a}$ is change event with
     subType ∈ {raise, lower, change parent}
3. objectID($O_i$) ≠ objectID($O_a$)
   and parent($O_i$) ≠ parent($O_a$)
   – $O_i$ and $O_a$ are change events with subTypes =
     change parent and target($O_i$) = target($O_a$)
   – $O_i$ and $O_a$ are change events with subTypes = set
     active
```

Fig. 3. Conflicting ADUs of the mlb

If no conflicting operation O_i is discovered, we can execute O_a immediately without a timewarp.

Finally, if there exists at least one conflicting O_i, it is checked if one O_i overwrites the effects of O_a, implying that the state of the medium after execution of O_a and O_i would have been identical to the state of the medium that has been reached by performing O_i only. If there is at least one such O_i, then O_a can be ignored, and no timewarp is necessary. As for determining conflicting operations, the application has to provide an appropriate function `overwrite(`O_i`, `O_a`)` which decides whether one operation may overwrite another operation.

Figure 4 specifies the overwrite function for the mlb. For example, let O_a and O_i be events that change the same attribute of an object (e.g, the color of a rectangle). Then the state of the object's attribute will reflect O_i after both O_a and O_i have benn executed. Even if a timewarp were executed, the user would miss the fact that the attribute of the object had a different value for a certain period of time, since only the state of the medium at T_C is displayed to the user.

```
overwrite($O_i$, $O_a$):
1. objectID($O_i$) = objectID($O_a$)
   – $O_i$ is state and $O_a$ is event
   – $O_i$ is delete event
   – $O_i \wedge O_a$ are change events
       · subType($O_i$) = subType($O_a$) excluding
           · objType ∈ {polygon, polyline}
             and subTypes = add point
           · objType = text and subTypes ∈ {change font,
             insert char, delete char, change cursor}
2. $O_i$ and $O_a$ are change events with subTypes =
   set active
```

Fig. 4. Overwriting ADUs of the mlb

It is interesting to note that the set of overwriting operations is a real subset of the set of conflicting operations, meaning that not all conflicting operations are overwriting as well. For example, let O_a be a state which indicated the creation of a new object on a whiteboard page and O_i be an event which changes the stacking order of another object on that page. Then O_a and O_i conflict regarding the display order of all objects belonging to that page, but O_i does not overwrite O_a.

This stepwise testing regarding the necessity of a timewarp is more difficult for continuous media due to the fact that here operations are valid only at their given timestamp. The execution of a late-arriving operation O_a generally requires a timewarp of the affected sub-component. Ignoring O_a is possible, but deciding whether the effect of O_a would have been completely overwritten seems to be more complex than in the discrete domain. In any case a continuous medium can still use the generic consistency service by instructing the service to skip the optimization.

C. Generic Timewarp Service

The improved timewarp has been implemented as a generic service. One task of the service is the management of the timewarp history. After operations have passed through the local lag service and have been executed by the application, they are handed over to the timewarp service. Since memory space is limited, operations cannot be stored infinitely, which has two implications: (1) the range of the timewarp is limited, meaning that an inconsistency caused by late-arriving O_a with T_a so that $T_C - T_a$ exceeds a certain threshold h cannot be repaired by timewarping. Extending h will increase the probability that an inconsistency can be repaired by a timewarp but will consume more memory space. The threshold can be chosen by the application in order to fine-tune this trade-off. For the mlb, h is set initially to 180 seconds and can be changed by the user. (2) The history does not start from the beginning of the session. A timewarp is therefore possible only if the history contains at least one state of each sub-component with a timestamp $\leq T_C - h$. Thus, the timewarp service requests every $\frac{h}{2}$ the state of the medium from the application (spread over a certain period of time to limit the application load). If a finer granularity is needed, the application can insert additional states into the history.

When the application receives a late-arriving operation, the generic service decides on the necessary actions as depicted in Figure 2. As mentioned above, tests regarding conflicting and overwriting operations are implemented by the application and are called via an appropriate interface function. Should a timewarp have to be performed, the service provides the application with a complete sequence of states and events in order to restore the medium's state. The application then executes this operation sequence like ADUs created or received normally.

VI. State Request

Even though the receipt of an operation outside the range of the timewarp history should be very unlikely, it cannot be ignored if guaranteed consistency is required. One reason for such a situation might be repeated packet loss and retransmission. If the local repair of an inconsistency is not possible, the affected application instance has to request the state of the inconsistent sub-component from other session members.

A similar problem occurs if session participants want to join an ongoing session. This requires that the late-joining application request the current state of the medium. Once an application has received an initial state for a sub-component it is able to maintain consistency by using the methods described above. In [VMG$^+$00] we describe a generic late-join service for RTP/I. This service provides selective and policy-based state initialization of the late-joining application. However, the late-join service does not ensure consistency when requesting the state of a sub-component. This is the task of a consistency service as presented here.

Consistency support for state requests raises two problems: first, which application instance should answer to a state request, and, second, how can it be guaranteed that the received state be consistent? The first problem is a typical example of feedback mechanisms where a request can be served by a number of session members, but only one answer (feedback) is needed. In order to prevent a so-called implosion of answers there exist several mechanisms to avoid feedback implosion [FW01]. We decided to use the state-of-the-art *exponential feedback raise* algorithm [NB99]. The basic idea is that each member able to serve the request set an exponentially distributed timer. If this timer expires, the state is sent. Should the answer of another application instance be received before the state is sent, the own timer is canceled, thus preventing multiple feedbacks.

The second problem is caused by the fact that an application which wants to answer a state request is not able to guarantee the consistency of the state it holds locally. For example, there might be a late-arriving operation en route to that application. In order to discover and repair this problem all applications check the transmitted states they receive against their local state copies. This can be easily done since the RTP/I framing identifies the events that are included in the transmitted state. Therefore, an application needs only to compare the information it holds locally with the information contained in the framing; it does not have to compare the actual states. The comparison can lead to four main results:

- The received state includes the same events as the state of the local application. In this case nothing has to be done. This is by far the most common case.
- The received state includes all events that are included in the state of the local application as well as some additional events. In this case the local application has missed some events and should adopt the received state.
- The state of the local application includes all events that are part of the received state as well as some additional events. In this case the remote application has sent an inconsistent state and the local application will send its own state to repair this problem (using the feedback suppression method explained above).
- Each state contains events that are not contained in the other one. In this case both states are inconsistent. Now it is checked which state contains more events, using the unique ID of the senders of the events as a tiebreaker if the number of events is equal. If the local state contains fewer events than the received state, then the local state is discarded and the received state is adopted. If the local state contains more events than the received state, then the local state is transmitted using the appropriate feedback suppression.

After a limited number of iterations this algorithm will result in all participants having the same state. If there is no single participant who has received all operations, the overall result is a consistent state across all participants which misses some events. This is acceptable since it can only happen because of the exceptional situation that a network partitioning has occured which lasted longer than the length of time the timewarp history is kept. In this situation it seems reasonable to keep the state of the partition where most changes have been executed and adapt the state of the other partition(s).

The generic consistency service can manage the state-request algorithm autonomously. The application needs to provide only functions for retrieving and setting the state of the sub-components.

VII. Intention Conflict Detection

The combination of local lag, timewarp, and state request is sufficient to ensure consistency according to the criterion defined in Section Three. However, even when this criterion is enforced, the original intention of a user may be violated. For example, consider two participants each changing the same attribute of a whiteboard object (almost) simultaneously. The total physical-time-ordering relation favors the participant whose operation takes place slightly later (or whose tie-breaker is greater). If the space of time between the two conflicting operations is very short, the effect of the operation created by the losing participant is not visible at all, leaving the user confused. To support collaboration it is therefore desirable to be able to identify intention conflicts and inform the users about them.

There exist a number of mechanisms to preserve user intentions that are realized together with consistency control. Operational transformation was originally designed for shared text editors; it transforms operations before their execution so that user intentions are maintained [SJZ$^+$98]. Another possibility would be to extend the total ordering relation so that in case of an intention conflict operations will be ordered according to certain priorities (e.g., the session chair wins over other participants). However, these mechanisms are very complex to develop and they are almost always medium-specific. We propose that intention conflicts should be made visible to the users so that they can resolve them and avoid further intention conflicts. For example, if two users try to move the same object on a shared whiteboard page, they should be informed that their intentions conflict. Typically they will then employ some social protocol to decide who will continue with the action.

The consistency service handles intention conflicts within the local lag queue and the timewarp history. The procedure is as follows: when receiving local and remote operations it is checked whether the operation causes an intention conflict involving the local user. Only operations which have timestamps that are within a certain threshold of the new operation's timestamp are considered for this purpose. This threshold is a parameter provided by the application. Preliminary experiments have shown that a value of around one second seems to be preferred by users of the mlb.

The decision whether or not two operations conflict with each other is made by means of a function provided by the application. Typically the conflict function described in the timewarp section can be reused for this purpose. In case a conflict is discovered, the application is provided with a list of all operations concerned. The application can then inform the users in an appropriate way. The mlb uses this mechanism to indicate an intention conflict by attaching a balloon help window to the affected objects that includes a list of rivaling participants (see Figure 5 for an example).

VIII. Experimental Evaluation

We evaluated our approach by conducting a series of experiments with a prototype of the multimedia lecture board. Our aim was to gain a first impression of how many timewarps can be prevented using the mechanisms described above and how much time is required for an individual timewarp. The performance of local lag and the discovery of intention violations was not evaluated in detail since their computational complexity is insignificant. This was confirmed by some preliminary experiments.

The experiments were conducted with 2 PCs equipped with Athlon 1000 processors, running Windows 98 as the operating system. The aim set for the two participants was to collaborate to create an outline of a protocol stack as shown in Figure 5.

In order to be able to state how much burden our approach places on an application we created a worst-case scenario for the experiments:

- An artificial network delay was introduced by buffering packets at the sender before transmitting them. This artificial delay was set to 150 ms, while the local lag was set to only 100 ms. Therefore, each operation arrived late and was a potential candidate for a timewarp.
- For the drawing of text we used poly lines rather than the regular text primitives. This was done since each mouse movement while drawing a poly line translates to one operation. Furthermore, each poly line segment is a separate object. Thus the number of operations and objects in the experiment were each very high.

The prime source of timewarps in this scenario is the creation of objects such as the poly line segments that are used for the text (see Figure 3). This is because upon the creation of an object the object is assigned a layer for the display order of the objects. An object which is created after another object is on a higher layer than the older object and may therefore obscure the older object if they intersect. Whenever the creation of two objects overlaps in time because of the artificial network delay, the creation operations are potential candidates for a timewarp, since the layer of one of the objects could be wrong. Another potential source of conflicting operations is the resizing and positioning of the boxes. When both participants are working on the same box, conflicting operations can happen.

We made three runs of the experiment. The results of these runs are shown in Table I. The first row identifies the run of the experiment. The second row shows the number of operations that were received late by a participant. Because of the artificial network delay this number is identical to the number of operations issued by the remote user. In a more natural environment with an adequate amount of local lag we expect that only those operations which are dropped by the network and have to be retransmitted will arrive late. It should be noted that the number of operations issued by the local user is not shown in the table. The number of operations was roughly equal for both users in all experiments.

The third column shows the number of operations that did not cause a timewarp since there was no operation with a greater timestamp when the remote operation arrived (see Figure 2). Column four shows the number of operations that did not cause a timewarp because they were not in conflict with operations that had a greater timestamp. The fifth column indicates the number of operations that were overwritten and therefore did not lead to a timewarp. Finally, the sixth column shows how many timewarps took place, while the seventh column displays the average amount of time required to perform a timewarp.

Overall it can be noted that only 0.3% to 1% of the late arriving operations caused a timewarp. Furthermore, the time required for a timewarp was only around 200 to 300 ms. Taking into account the large number of operations that had to be executed during the timewarp and the duration of the experiments (3-5 minutes), this seems very acceptable. Also, while conducting the experiments we discovered some inefficiencies with the maintenance of the queue for the operations that are stored to perform a timewarp. We expect that improving the implementation will cut the time required for the timewarp by at least 50%.

IX. Related Work

In the last decade, much work has been done in the area of optimistic consistency control for replicated applications. One of the prime existing approaches is operational transformation [SJZ+98], [EG89]. With operational transformation remote operations are transformed before their execution so that the application's state is consistent after execution of the transformed operation (in terms of convergence, causality preservation, and intention preservation). Originally, operational transformation was designed for shared text editors. Designing the correct transforming functionality for more complex applications is considered very difficult, especially in the case of continuous media. In contrast, it is fairly straight-forward to realize local lag and timewarp for an application.

An alternative approach to the total physical-time-ordering relation is causal ordering as defined by Sun/Ellis [SE98] derived from Lamport's work on physical clocks [Lam78]. However, causal ordering is not feasible for continuous media since operations need additionally to be executed at the correct point in time. In contrast, local lag, timewarp, and state request can be applied to both the discrete and the continuous domains.

Another example of optimistic consistency management is object replication [SC00]. The idea is to handle intention conflicts due to simultaneous operations changing the same objects by replicating the object concerned. Instead of finding a total order between all rival operations and showing only the effects of the last operation, multiple versions of the same object are created and displayed. The participants can either select a certain version or keep them all.

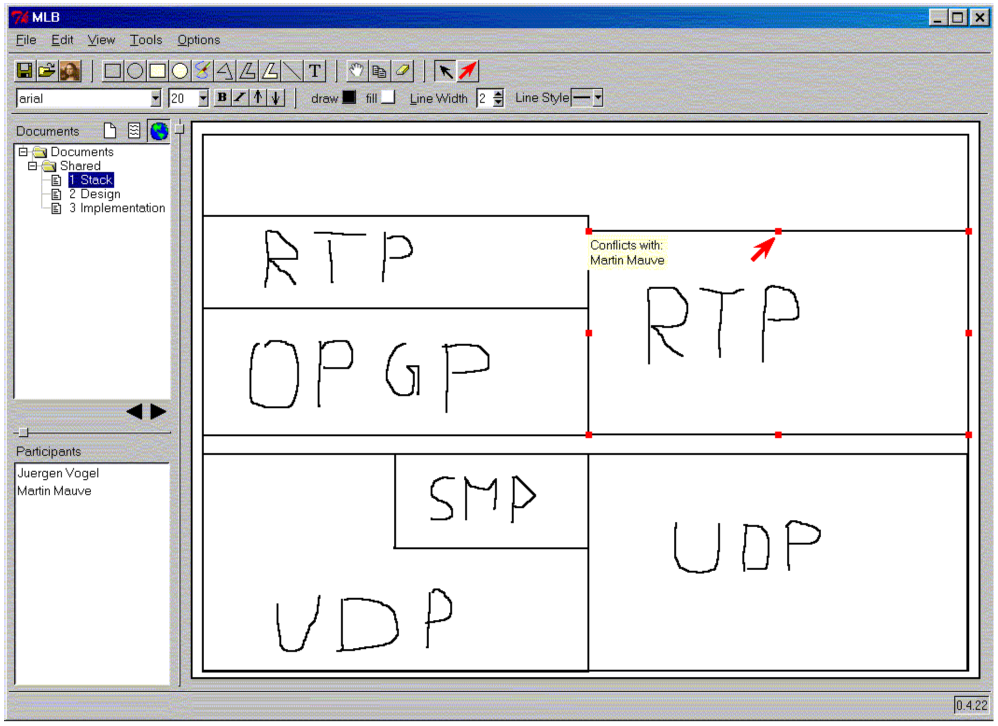

Fig. 5. mlb Screenshot

TABLE I
Timewarp Performance

run	late operations	no later operations	no conflict	overwrite	timewarps	average timewarp time
1	687	310	323	47	7	281 ms
2	594	246	314	31	2	250 ms
3	515	193	295	25	2	195 ms

Drawbacks to this approach are a complex management of multiple versions of the same object, and a confusing effect for the user if too many different versions exist (e.g., when subsequent operations create successive versions). Furthermore for continuous interactive media this approach seems to be problematic. For example, consider a networked computer game in which object duplication would result in two representations of the same object in the game. The combination of local lag and timewarp prevents this problem and makes sure that the end result is the same as if all user operations had been executed in the correct order at the correct point in time.

In [Edw97] an approach to the handling of intention conflicts in collaborative applications is presented. This work focuses on detection and resolution of conflicts that violate the application state integrity (e.g., moving an object after it has been deleted). Conflict detection is done within the operation history, which is similar to our approach. However, we rely on the users to resolve intention conflicts in order to provide a light-weight service which is application-independent.

In [Mau00] we explored the theoretical background of local lag and (basic) timewarp in continuous media. The work presented here improves on these concepts and shows how they can be optimized to also be useful in the discrete domain. Furthermore, we now provide a generic service which allows arbitrary RTP/I-based applications to integrate these mechanisms. In the present paper we also evaluate the performance of the improved timewarp and we extend the consistency support to include mechanisms for the detection of intention conflicts and consistent state requests. The latter is required in particular to support late-joining participants.

X. Conclusion and Outlook

We have presented an optimistic consistency service for distributed interactive media. The service eliminates a large percentage of inconsistencies by voluntarily delaying local operations. This method is called local lag. Those inconsistencies that cannot be prevented are repaired using an improved timewarp algorithm. Timewarp can repair inconsistencies using exclusively local information without burdening the network or the applications of other users. Our advanced timewarp algorithm improves upon existing timewarp approaches by minimizing the number of required timewarps. For handling exceptional situations and in order to support late-join functionality, the consistency service is able to request parts of the medium's state in a consistent way. Finally, the consistency service is able to detect intention conflicts between the actions of different users.

The consistency service has been implemented in C++ as a generic service which is based on the application-level protocol for distributed interactive media (RTP/I). Any application using RTP/I can reuse the generic service without further modification. In order to demonstrate the feasibility of our approach we have integrated the consistency service into the multimedia lecture board (mlb), a shared whiteboard system. To test the performance of the consistency service we have conducted a preliminary evaluation of the advanced timewarp algorithm. The results show that a significant number of timewarps can be prevented in the discrete domain and that the amount of computational resources required for the improved timewarp is quite acceptable.

In the future we will conduct a real-life evaluation of the service by using the mlb during teleseminars between several European universities. Furthermore, we plan to integrate the consistency service into other applications for distributed interactive media. Of particular interest will be the use of the service for continuous media like networked computer games or distributed virtual environments. We expect that the main challenge will be the definition of the appropriate functions (such as conflict and overwrite) that have to be provided by the application.

Acknowledgments

Development of the multimedia lecture board is part of the project ANETTE, which is supported by the DFN-Verein under contract TK602-VA/T 102.1.

References

[Edw97] W.K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *ACM UIST*, pages 139–148, 1997.

[EG89] C.A. Ellis and S.J. Gibbs. Concurrency Control in Groupware Systems. In *ACM SIGMOD*, pages 399–407, 1989.

[FW01] T. Fuhrmann and J. Widmer. On the Scaling of Feedback Algorithms for Very Large Multicast Groups. *To appear in: Special Issue of Computer Communications on Integrating Multicast into the Internet*, 2001.

[GD98] L. Gautier and C. Diot. Design and Evaluation of MiMaze, a Multi-player Game on the Internet. In *IEEE International Conference on Multimedia Computing and Systems*, pages 233–236, 1998.

[GE98] W. Geyer and W. Effelsberg. The Digital Lecture Board — A Teaching and Learning Tool for Remote Instruction in Higher Education. In *Proc. ED-MEDIA/ED-TELECOM'98*. AACE Association for the Advancement of Computing in Education, 6 1998. [on CD-ROM only].

[Hag96] O. Hagesand. Interactive multiuser VEs in the DIVE system. *IEEE Multimedia*, 3(1):30–39, 1996.

[HC97] M. Handley and J. Crowcroft. Network Text Editor (NTE) – A scalable shared text editor for the MBone. In *ACM SIGCOMM*, pages 197–208, 1997.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Mau00] M. Mauve. Consistency in Continuous Distributed Interactive Media. In *ACM CSCW*, pages 181–190, 2000.

[MD96] J. Munson and P. Dewan. A Concurrency Control Framework for Collaborative Systems. In *ACM CSCW*, pages 278–287, 1996.

[MH00] M. Mauve and V. Hilt. An Application Developer's Perspective on Reliable Multicast for Distributed Interactive Media. *ACM Computer Communication Review*, 30(3):28–38, 2000.

[MHKE01] M. Mauve, V. Hilt, C. Kuhmñch, and W. Effelsberg. RTP/I - Toward a Common Application-Level Protocol for Distributed Interactive Media. *IEEE Transactions on Multimedia*, 3(1), 2001.

[Mil92] D.L. Mills. Network Time Protocol (Version 3) specification, implementation and analysis. DARPA Network Working Group Report RFC-1305, University of Delaware, 1992.

[NB99] J. Nonnenmacher and E. W. Biersack. Scalable feedback for large groups. *IEEE/ACM Transactions on Networking*, 7(3):375 – 386, June 1999.

[SC00] C. Sun and D. Chen. A Multi-version Approach to Conflict Resolution in Distributed Groupware Systems. In *Proc. of the 20th IEEE Interaction Conference on Distributed Computing Systems*, pages 316–325, 2000.

[SCFJ99] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Internet draft, IETF, Audio/Video Transport Working Group, draft-ietf-avt-rtp-new-03, February 1999. Expiring date August 26^{th}, 1999.

[SE98] C. Sun and C. Ellis. Operational transformation in real-time group editors:issues algorithms, and achievements. In *Proc. of the ACM CSCW*, pages 59–68, 1998.

[SJZ$^+$98] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality Preservation and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.

[Tun98] T.L. Tung. MediaBoard. Master's thesis, University of California, Berkely, California, USA, 1998.

[VGB99] I. Vaghi, C. Greenhalgh, and S. Benford. Coping with Inconsistency due to Network Delays in Collaborative Virtual Environments. In *ACM VRST*, pages 42–49, 1999.

[VMG$^+$00] J. Vogel, M. Mauve, W. Geyer, V. Hilt, and C. Kuhmmünch. A Generic Late Join Service for Distributed Interactive Media. In *8th ACM Multimedia, ACM MM 2000*, pages 259–268, 2000.

[Vog01a] J. Vogel. multimedia lecture board (mlb). URL: http://www.www.informatik.uni-mannheim.de/informatik/pi4/projects/ANETTE/anetteProject2.html, 2001.

[Vog01b] J. Vogel. RTP/I Payload Type Definition for Shared Whiteboards. Technical Report 5/2001, Department for Praktische Informatik IV, University of Mannheim, Germany, 2001.