Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications

Martin Mauve[1], Jürgen Vogel[1], Volker Hilt[2] and Wolfgang Effelsberg[1]

[1]Praktische Informatik IV
University of Mannheim
L15, 16
68161 Mannheim
Germany
Phone: +49 621 181 2616
Fax: +49 621 181 2601
e-mail: {mauve,vogel,effelsberg}@informatik.uni-mannheim.de

[2]Bell Laboratories
101 Crawfords Corner Rd
Holmdel, NJ 07733
USA
Phone: +1 732 332 6432
e-mail: volkerh@dnrc.bell-labs.com
The work described in this paper was conducted while Volker Hilt was with the Universität Mannheim.

Abstract— **In this paper we investigate how consistency can be established for replicated applications changing their state in reaction to user-initiated operations as well as the passing of time. Typical examples of these applications are networked computer games and distributed virtual environments. We give a formal definition of the terms *consistency* and *correctness* for this application class. Based on these definitions, it is shown that an important tradeoff relationship exists between the responsiveness of the application and the appearance of short-term inconsistencies. We propose to exploit the knowledge of this tradeoff by voluntarily decreasing the responsiveness of the application in order to eliminate short-term inconsistencies. This concept is called local-lag. Furthermore, a timewarp scheme is presented that complements local-lag by guaranteeing consistency and correctness for replicated continuous applications. The computational complexity of the timewarp algorithm is determined in theory and practice by examining a simple networked computer game. The timewarp scheme is then compared to the well-known dead-reckoning approach. It is shown that the choice between both schemes is application-dependent.**

Index Terms— **EDICS: 3-VRAR, 6-MMMR, 7-CONF, additional keywords: Networked Games, Consistency, Replicated Continuous Applications, Timewarp, Local-Lag**

I. Introduction

CONSISTENCY in replicated applications and in distributed systems has received extensive attention in theory and practice. The vast majority of work in this area has been carried out with the assumption that the applications are discrete and change their state only in response to (user initiated) operations. Examples for this class of applications are networked text editors [1], [2] and shared drawing tools [3]. We refer to this class as the *discrete domain*.

However, currently a large number of replicated applications evolve that change their state not only in response to operations, but also because of the passing of time. Very prominent examples for these applications are networked computer games [4]. Other examples are shared virtual reality systems [5], as well as CSCW applications for joint work with dynamic objects [6]. We call these applications the *continuous domain*. For this class of applications the issue of consistency is still largely unexplored. As we shall show, the approaches for replicated discrete applications are not usable in the continuous domain since they ignore state changes caused by the passage of time.

In the area of distributed virtual environments (DVEs) [5], [7], [8] a mechanism called dead-reckoning has been specifically developed to tackle the problem of consistency in the continuous domain. While dead-reckoning can pro-

vide consistency in the continuous domain, we will demonstrate that it does not prevent a consistent but incorrect state. While this might be reasonable for battlefield simulations for which dead-reckoning was originally designed, it is not suitable for other replicated continuous applications. To address this problem, it is necessary to investigate the issue of consistency and correctness in the continuous domain in a more general way than this has been done before.

An essential part of this work is therefore dedicated to the formal specification of a consistency and a correctness criterion for the continuous domain. These definitions allow the identification and evaluation of an important tradeoff between responsiveness and the appearance of brief periods of inconsistency, called short-term inconsistencies. This leads to the introduction of the concept of *local-lag*: deliberately decrease responsiveness to lower the number and to reduce the duration of short-term inconsistencies.

While local-lag makes the appearance of short-term inconsistencies less likely, it cannot completely prevent them. We therefore propose to use a timewarp algorithm to repair short-term inconsistencies when they occur. The timewarp algorithm provides consistency and correctness.

As a proof that local-lag and timewarp are viable concepts, we have developed a simple networked computer game. The game is used to verify the theoretical observations of the algorithms for local-lag and timewarp.

The remainder of this paper is structured as follows: Section Two introduces the basic terminology that we use to discuss consistency in the continuous domain. It is shown in Section Three that mechanisms to ensure consistency in the discrete domain cannot be used for the continuous domain. In Section Four, the terms consistency and correctness in the continuous domain are formally defined, and the tradeoff between responsiveness and short-term inconsistencies is identified. The concepts of local-lag and timewarp are presented in Sections Five and Six, respectively. Section Seven contains the discussion of a simple networked computer game, verifying the theoretical results from the previous sections. Dead-reckoning and timewarp are compared in Section Eight. Section Nine concludes the paper.

II. Terminology

The key characteristic of a *replicated application* is that a local copy of the application's state is maintained simultaneously by multiple *application instances*. These application instances may reside on different computers that are connected by a computer network, they are also termed *sites*.

The state of a replicated application may change because of (user-initiated) *operations*. An operation may be issued at an arbitrary site. The operation or its effect on the state of the replicated application needs to be communicated to all sites so that all local state copies can be updated accordingly.

We call a replicated application *discrete* if it changes its state only in response to (user-initiated) operations. In *continuous* replicated applications the state may also change because of the passage of time without requiring the exchange of information. This implies that each instance of the application has access to a physical clock which can be used to measure the progress of time. In a real world system, these physical clocks will never be fully synchronized. In the remainder of this work we use the term *time* to refer to one specific reading of one specific physical clock. Due to limited synchronization this reading may not be reached simultaneously (in the sense of a common wall-clock) by all physical clocks.

In order to simplify the further discussion, we denote with $s_{i,t}$ the state that site i holds at time t. Likewise o_{i,t^o,t^*} identifies an operation that has been issued at site i at a time t^o, with t^* being the time the operation is supposed to be executed. For now we assume that $t^o = t^*$. We shall discuss later that it may make sense to set t^* to a value greater than t^o. It is assumed that the resolution of the physical clock is sufficiently high so that no two operations can be issued with the same execution time t^* at the same site i, and therefore an operation is uniquely identified by i and t^*. If this is not the case, an additional local counter can be used to distinguish operations with identical values for i and t^*. The set of all operations o_{i,t^o,t^*} that occur during the lifetime of a continuous replicated application is called O.

III. Consistency in the Discrete Domain

Existing approaches such as [9], [1], [2] to establish consistency in the discrete domain are about finding a 'correct' sequence of all those operations that are not independent of each other. These algorithms make sure that eventually, at every site, the state looks as if all dependent operations, issued at all sites, had been executed successfully in that particular sequential order. The common root of these approaches is Lamport's work on virtual clocks [10].

With the tremendous amount of earlier work invested into consistency for replicated discrete applications, the question arises whether these approaches can also be reused in the continuous domain. As we shall show, this is not the case. The reason why the approaches for the discrete domain fail when they are applied to the continuous domain is that consistency in replicated continuous applications is not only about finding a correct sequence of operations and ensuring that at each site the result of all operations looks as if the operations had been executed in that sequence. In addition it requires that the result of all operations looks as if the operations had been executed at the *correct point in time*. The algorithms for establishing consistency in the discrete domain are therefore insufficient for the continuous domain.

To illustrate this problem, we examine a very simple example, based on a session involving a replicated discrete application. This session is attended by two users U_1 and U_2. U_1 performs an operation. This operation will be executed first by U_1's instance of the application and some time later (because of the network transmission delay) by U_2's application instance. Since for a replicated discrete application a single operation cannot result in inconsisten-

cies, consistency algorithms from the discrete domain will take no special action in this example.

Now we transfer this example to the continuous domain: in a distributed simulation a train is approaching a switch. The simulation is attended by two users, U_1 and U_2. Just before the train arrives at the junction, U_1 operates the switch. In U_1's application instance the operation takes place immediately. However, the information about U_1 operating the switch will arrive at U_2's application instance at a later point in time, due to network delay. Applying the operation at this point in time to U_2's application instance leads to an inconsistent state because the train has already passed the switch in U_2's application instance. As explained above, methods for ensuring consistency in the discrete domain will take no action to correct this problem. This reveals the core reason why the mechanisms for consistency used in the discrete domain are not sufficient for continuous replicated applications: they neglect the problem of executing operations at the correct point in time.

IV. Consistency in the Continuous Domain

In the following the terms *consistency* and *correctness* for the continuous domain will be defined and it will be shown that the synchronization of the physical clocks of the individual sites has *no* impact on both aspects. In addition, an important tradeoff between consistency-related artifacts for this application class will be identified which *is* influenced by the synchronization of the participating site's physical clocks.

A. Consistency and Correctness

Let the reception function R_i be defined to return *false* if a given operation has been received by site i after time t and *true* otherwise (see Equation 1).

$$R_i(t, o_{j,t^o,t^*}) = \begin{cases} false & o_{j,t^o,t^*} \text{ received by i after } t \\ true & \text{else} \end{cases} \quad (1)$$

Then we define the term consistency for the continuous domain by means of a consistency criterion. *A replicated continuous application ensures consistency iff Expression 2 evaluates to true.* The interpretation of this expression is as follows: at any time t the state at any two sites i and j must be the same, if both sites have received all operations that are supposed to be executed before t.

$$\begin{aligned} &\forall t, i, j \mid \forall t^* \leq t,\ o_{w,t^o,t^*} \in O \mid \\ &R_i(t, o_{w,t^o,t^*}) \wedge R_j(t, o_{w,t^o,t^*}) \Rightarrow (s_{i,t} = s_{j,t}) \end{aligned} \quad (2)$$

This definition implies two important things. First, sites which have not yet received all information required to calculate the correct state do not affect the question whether the application ensures consistency or not. This is important since otherwise in the presence of network delay, no application would ever be able to provide consistency. Second, if at any time t all sites have received all operations, then the state at all sites at that time must be identical. This ensures the common meaning of the term consistency.

The consistency criterion given here is a stronger requirement when compared to those typically used in the discrete domain (e.g., [2]). By requiring that $s_{i,t} = s_{j,t}$ it takes into account that the state may change over time, which is not considered (and not necessary) in the discrete domain. The consistency criterion for the continuous domain can thus be regarded as a specialization of the consistency criteria used in the discrete domain.

It should be noted that consistency is completely independent of the synchronization of the distinct physical clocks. The consistency criterion only requires that at the same reading of the physical clocks the same state is reached if all preceding operations have been received. It is irrelevant whether or not the same state is reached at the same reading of a common wall-clock.

Besides consistency one might also require that the state of a replicated continuous application be *correct* in the sense that it is the state that would have been reached by issuing and executing all operations on one single instance of the application at the correct point in time. Guaranteeing consistency only makes sure that all sites determine the same state provided that they have received all operations that are required to do so. This is not necessarily the state that would have been reached by executing every operation o_{i,t^o,t^*} at t^*.

In order to provide a formalized definition of the term correctness we define a virtual perfect site P such that:

- for any operation $o_{i,t^o,t^*} \in O$ $R_P(t^*, o_{i,t^o,t^*}) = true$, i.e., all operations are received by P by the time they should be executed, and
- P calculates the state of the application by executing all operations $o_{i,t^o,t^*} \in O$ at t^* and by determining all time-dependent state changes according to the rules of the application.

The virtual perfect site P therefore always has the state that a single non-distributed application would have when fed with the same operations.

Correctness can then be defined by the following correctness criterion: *a replicated continuous application provides consistency and correctness, iff Expression 3 evaluates to true.* The interpretation of this expression is as follows: at any time t the state at any site i must be the same as the state of the virtual perfect site P, provided that i has received all operations that are supposed to be executed before t. Note that Expression 3 implies consistency. Furthermore, as for consistency, correctness is completely independent of the synchronization of the physical clocks at distinct sites.

$$\begin{aligned} &\forall t, i \mid \forall t^* \leq t,\ o_{w,t^o,t^*} \in O \mid \\ &R_i(t, o_{w,t^o,t^*}) \Rightarrow (s_{i,t} = s_{P,t}) \end{aligned} \quad (3)$$

Strictly speaking the definitions of consistency and correctness require that the state can be calculated without consuming time, i.e., that the algorithms used to establish consistency or correctness are executed arbitrarily fast.

This is acceptable for a theoretical discussion of algorithms. In a real world system, the algorithms for achieving consistency or correctness will require time to calculate a new state. However, due to the limited resolution of the human perception this does not pose a real problem as long as the state calculation is sufficiently fast.

Figure 1 illustrates the terms consistency and correctness. In this example, one separate instance of a continuous distributed application is running at each (real) site 1 and 2. These sites are connected by a computer network. P is the virtual perfect site which receives all operations such that they can be executed at t^*. Sites 1 and 2 issue one operation with $t^o = t^*$. Provided that the clocks of all sites are perfectly synchronized and assuming that the transmission over the computer network takes a positive amount of time, both operations will arrive late (i.e., after t^*) at the site that did not issue it. Following our definition, the application is *consistent* if it guarantees that for $t < 2$ and $t \geq 6$ the states at both sites are identical. *Correctness* is achieved if the state of site 1 is identical to the state of P for $t < 3$ and $t \geq 6$ and if the state at site 2 is identical to the state of P for $t < 2$ and $t \geq 5$.

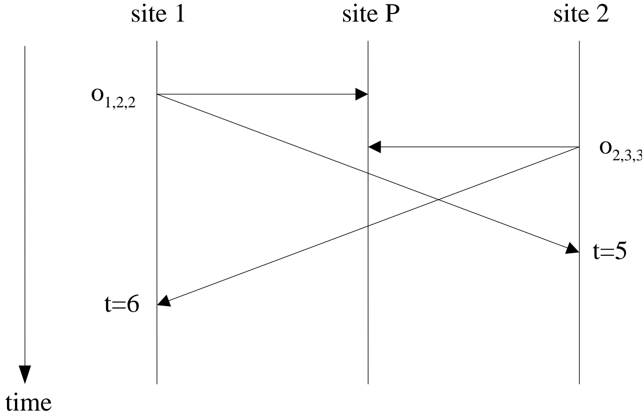

Fig. 1. Consistency and correctness

B. Short-Term Inconsistencies and Responsiveness

Both the consistency and correctness criteria make no statements about the time during which a site has not yet received all operations that it needs to calculate the current state. It is therefore possible that consistency-related, transient artifacts occur, even if the correctness criterion is met by the application. In the example above, this could happen during $3 \leq t < 6$ for site 1 and during $2 \leq t < 5$ for site 2. In addition to the basic consistency and correctness criteria, we therefore define two additional fidelity criteria.

The first fidelity criterion is the avoidance of short-term inconsistencies. *If for an operation o_{i,t^o,t^*} and a site j the expression $R_j(t^*, o_{i,t^o,t^*})$ evaluates to false, then this operation is said to have caused a short-term inconsistency of the application at site j.* Ideally, short-term inconsistencies should be non-existent.

It should be noted that the term 'short-term inconsistency' identifies a situation where the state of at least two sites is different from each other because an operation issued at site i arrives at site j after t^*. It does not refer to a situation where the consistency or the correctness criterion are violated. Because we assume that the application ensures the consistency or the correctness criterion, a late arrival means that the state of at least one site needs to be repaired. In the previous train example this would mean moving the train from a position on one branch of the tracks to a different position on the other branch of the tracks for one of the participants. This results in a consistency-related artifact that may be visible to a user.

Furthermore, operations may be issued while at least one site is encountering a short-term inconsistency. It may therefore happen that an operation is based on a state that needs to be corrected by the employed consistency or correctness mechanisms. For example, assume that in the train example U_2 pulls the brakes of the train after it has passed the switch but before the operation from U_1 has arrived. In this case, U_2 intends to stop the train because it is going in the 'wrong' direction. After U_2 has issued this operation, the operation from U_1 arrives (late). If the employed consistency mechanism modifies the state at the site of U_2, then the operation of stopping the train would cause the train to be stopped while heading in the 'right' direction. This might violate the original intention of U_2. Operations that are based on a state visible during a short-term inconsistency are therefore another source for consistency-related artifacts.

The length of a short-term inconsistency depends on the offset between the physical clocks, the transmission delay and on how much earlier an operation was issued (t^o) than it should be executed (t^*). It can be calculated for site j and operation o_{i,t^o,t^*} as shown in (4), where T_k^* denotes the reading of a common wall clock at the time the physical clock at site k reaches t^* and $d_{i,j}$ denotes the (unidirectional) network delay from i to j. If (4) yields a negative result then the operation has not caused a short-term inconsistency. We assume that the longer a short-term inconsistency persists, the more likely it is that the implications discussed above will distract the users.

$$I_j(o_{i,t^o,t^*}) = d_{i,j} + T_i^* - T_j^* - (t^* - t^o) \qquad (4)$$

The second fidelity criterion concerns the response time of a continuous replicated application. *The response time of an operation o_{i,t^o,t^*} is defined as $t^* - t^o$.* The ideal response time is zero.

If the response time exceeds a certain threshold, the users will notice that a delay exists between the time the operation was issued and the time the operation is executed. This will result in an unnatural behavior of the application. Response time and responsiveness are used synonymously in the context of this work.

While it would be desirable to guarantee that no consistency-related artifacts occur in a continuous replicated application, this is not possible. The reason is that the optimization of the response time and the avoidance of short-term inconsistencies are conflicting goals. The connection between both fidelity criteria is contained in Equa-

tion 4 where the response time can be increased to decrease the duration of a short-term inconsistency (and vice versa).

To illustrate this tradeoff, Figure 2 shows an example where the responsiveness is optimal ($t^o = t^*$). In this case the operations issued by the users take effect immediately. The points in time when the two operations should take effect are indicated by the dotted line. As can be seen, only the user issuing the operation does not experience short-term inconsistencies. Any other user will encounter a short-term inconsistency for each operation due to the transmission delay in the network.

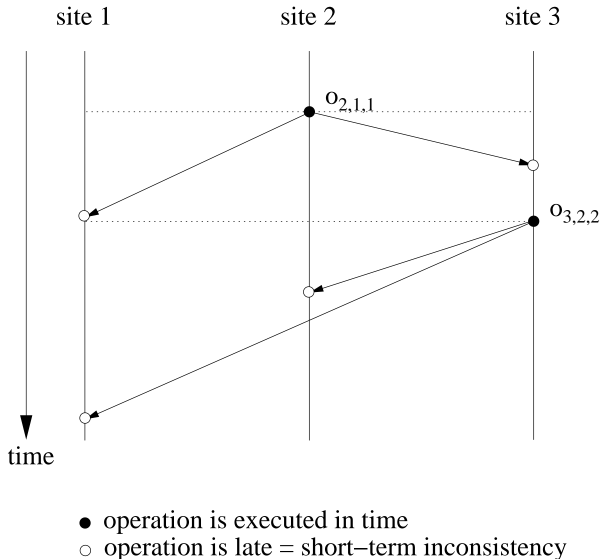

Fig. 2. Optimizing responsiveness

Figure 3 shows the same example optimized for the short-term inconsistency criterion. Here short-term inconsistencies are completely avoided at the cost of increasing the response time for each operation to the maximum transmission delay (and possible deviations in the physical clocks) between any two participants. Note that the execution of the operations is delayed at the originating sites until all participants have a chance to act simultaneously. An event received at a time before it should be executed does not pose a problem. It is buffered by the receiver until the time denoted by t^* is reached.

There are two additional questions that remain to be answered: What if the deviation in physical clocks between sender and receiver happens to compensate for the transmission delay? And, in an environment where network packets might get lost, how exactly can we calculate the maximum transmission delay for an operation?

The answer to the first question is that the time deviation might indeed compensate for the network delay between a given sender and receiver. This makes it possible for the sender to have a response time equal to zero while the receiver does not experience short-term inconsistencies. However, this works only when the sender and the receiver do not switch their roles. As soon as the previous receiver becomes the sender, short-term inconsistencies will occur

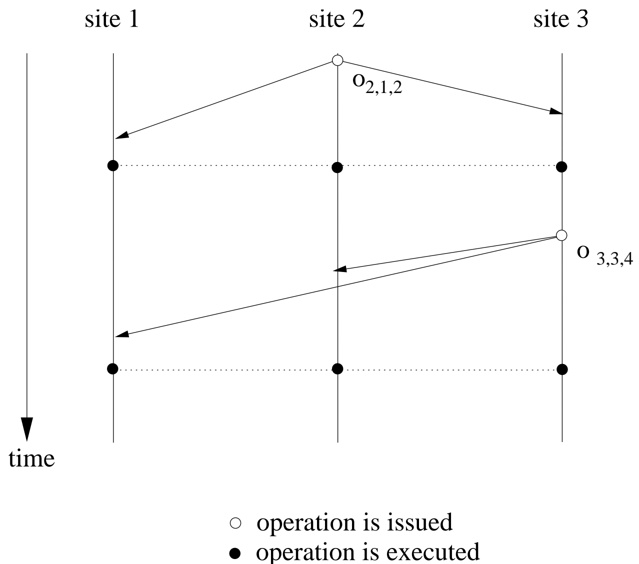

Fig. 3. Minimizing short-term inconsistencies

because of the large time deviation which now prevents that any operation arrives at the new receiver in time.

The second observation arises from the problem that in an environment where packet loss occurs, it is impossible to define an upper bound on the delay an operation needs to arrive at a remote site. After all, the same packet might get lost over and over again as it is being retransmitted by the sender. This leads to the conclusion that, while a reduction of short-term inconsistencies is desirable, a guaranteed prevention is not possible in networks that do not guarantee the delivery of packets within a certain amount of time.

C. Causality Preservation

In the discrete domain, consistency often includes causality preservation. That is, if there are two operations O_1 and O_2 that are not independent of each other (e.g., because the user issuing operation O_2 has seen the result of O_1 before issuing O_2), then it must be guaranteed that both operations are executed in the correct order at each site. This is done by delaying the local execution of operations until all operations they depend on have been executed.

In the continuous domain, this poses an interesting problem: as long as there are no short-term inconsistencies, causality is preserved since the operations are executed at the correct point in time. Only when operations arrive late and cause a short-term inconsistency it is possible that one operation 'overtakes' an operation it depends on. If causality preservation should be guaranteed in this case, it would require the delay of the operation until the operations it depends on have arrived and have been executed. However, delaying the operation past the point where it should be executed would cause another short-term inconsistency, which is undesirable.

For this reason, causality preservation in not included in the consistency or correctness criterion for the contin-

uous domain. It is implicitly contained in the short-term inconsistency fidelity criterion.

V. Local-Lag

In order to make use of the trade-off relationship between response time and short-term inconsistencies, we propose a concept called *local-lag*: Instead of immediately executing an operation issued by a local user, the operation is delayed for a certain amount of time before it is executed, i.e., $t^* > t^o$ for an operation o_{i,t^o,t^*}. As depicted in Figure 3, if the value for local lag is sufficiently high, it can reduce the number and/or duration of short-term inconsistencies.

In the example with the train approaching the switch, local-lag would work as follows: user U_1 would operate the switch. This operation would be assigned $t^* > t^o$. If the difference between t^* and t^o is sufficiently large, this would mean that the site of user U_2 would receive the operation before it is due for execution at t^*. Thus, the short-term inconsistency would be prevented. On the other hand, the execution of this operation for U_1 would be delayed. If this delay is too high, it would be noticeable or even distracting for U_1, e.g., the user issues the operation a long time before the train passes the switch but the operation takes effect some time after the train has passed it. Thus, it is important that local-lag is chosen in a way such that is not noticeable or at least not distracting for the local user.

It should be noted that the idea of delaying local operations is not new. For example, in [4] local state updates are delayed by 100 ms for a networked computer game. The key contribution of our concept of local-lag is that the value for the delay is not arbitrarily set but is based on the knowledge about the tradeoff between short-term inconsistencies and response time.

A. Determining a Value for Local-Lag

As discussed above, it would not be desirable to move from one extreme, where the response time is zero but short inconsistencies are very frequent, to the opposite extreme, where almost no short-term inconsistencies occur but the response time is unacceptably high. Therefore, it is important to consider both consistency-related fidelity criteria. For a given replicated continuous application, we propose to do this in three steps: (1) determine a minimum for the local-lag needed to prevent a significant amount of short-term inconsistencies, then (2) determine the highest acceptable response time, and finally (3) choose a value for the local-lag as a compromise.

A.1 Determining a minimal value for local-lag

In order for local-lag to be useful, it needs to significantly reduce the number of short-term inconsistencies for all participants. Moreover, short-term inconsistencies should be reduced for each sender/receiver pair. Therefore, the local-lag should be chosen such that there is a high probability that $I_j(o_{i,t^o,t^*})$ will evaluate to a value which is equal to or less than 0 for any site j and any operation o_{i,t^o,t^*}.

We therefore propose to use the maximum of the average network delays $d_{i,j}$ between any two sites i and j as the minimum amount of local-lag. Typical values for network delays (assuming an uncongested network) are: less than 1 ms for a LAN, 20 ms within a European country, 40 ms within a continent, and 150 ms for a world-wide session. If the clocks of the participants are not completely synchronized, the maximum offset between any two clocks should be added to this value. Choosing the maximum of the average network layer delays adjusted by the clock offset as minimal value for local-lag implies that short-term inconsistencies will occur only if packets get lost or the delay jitter becomes significant because of network congestion.

A.2 Determining the highest acceptable response time

The maximum local-lag is dependent on how much response time a user can tolerate for a given application. In order to determine this value, the work conducted in the area of System Response Time can provide a good orientation [11], [12], [13]. Furthermore, for a given specific application it may be a good idea to conduct perceptual psychological experiments. Ideally, the experiments will deliver the value of local-lag that is not noticeable by the user. The literature about System Response Time suggests that a lag value of 80-100 ms will not be noticeable by the user, independent of the operation and the application. Our own experiments (described later), indicate that this value may be larger for certain applications.

A.3 Choosing a value for the local-lag

The previous two steps can result in two main cases: either the minimum value for local-lag is smaller than or equal to the highest acceptable response time, or it is not. In the first case a value between the minimum value for local-lag and the highest acceptable response time can be chosen. If the highest acceptable response time is lower than the minimal useful value for local-lag, then a true tradeoff situation occurs. Given the values mentioned above, this should be relatively rare, though it might happen for very demanding applications. In this case it is necessary to lower the fidelity of both criteria in a way which provides the best overall fidelity to the user. Most likely this will require another set of perceptual psychological experiments.

VI. Timewarp

If the amount of local-lag is sufficiently large, it eliminates a significant amount of short-term inconsistencies. However, it does not completely prevent all inconsistencies since operations might still arrive late due to jitter or packet loss in the network. In order to ensure consistency or correctness, it is therefore necessary to have a mechanism to repair the state in these cases. In this section we propose such a mechanism which we call timewarp.

A. Algorithm

We assume that each application instance i is initialized at a potentially different point in time t_i^I with the correct state $s_{i,t_i^I} = s_{P,t_i^I}$. Timewarp requires that each application instance maintains a list of operations L_i^o and a list

of states L_i^s. The lists are assumed to be sorted by t^* and t, respectively. L_i^o is initialized to be empty and L_i^s is initialized to contain only s_{i,t_i^I}. With these preconditions the timewarp algorithm can be defined as follows:

- *Step 1.* Wait for a constant amount of time T. While waiting receive local and remote operations and store them in L_i^o, and remember t_w^o as the smallest t^* of any operation received during T. T determines the frequency with which new states are calculated and displayed to the user. For example, if 20 updates per second must be shown to the user, then T should be set to 50 ms minus the estimated time required for the other steps.
- *Step 2.* Determine t_w^s such that $t_w^s = max\{t|(s_{i,t} \in L_i^s) \wedge (t < t_w^o)\}$. I.e., choose the state in the list of recorded states that directly precedes the earliest operation received in step 1. t_w^s denotes the time this state was valid.
- *Step 3.* Let t^c be the current time. Do the following for all times t with $t_w^s < t < t^c$ where there exists at least one operation $o_{k,t^o,t} \in L_i^o$ in ascending order. Take s_{i,t_w^s} if this is the first iteration, the state resulting from the previous iteration otherwise. This is called the base state s_{i,t^b} of this iteration. Using s_{i,t^b} as a starting point calculate the state at time t according to the rules of the application. While doing this recalculate and replace each $s_{i,n} \in L_i^s$ with $t^b \leq n < t$. Execute the operations with $t^* = t$ on the state calculated for t according to the rules of the application. Continue with the next iteration. The informal description of this step is as follows: starting with the state selected in step 2 all operations that happened after that state was valid are applied to that state in fast forward mode. This is done by determining the state before each operation should have been executed and then executing it. States in the list of states are replaced by updated versions that take the events into account that have arrived during step 1.
- *Step 4.* Use the resulting state from step 3 to calculate the state at the current time t^c, save that state to L_i^s and display it to the user. Go to step 1.

In the train example timewarp would work as follows: the site of user U_2 would receive the operation after t^* during step 1. This can happen, e.g., because the applied amount of local-lag was not sufficient to compensate for the network delay. In Step 2 the last recorded state preceding t^* would be reconstructed. From this state on all events would be applied to that state in fast forward mode during Step 3, including the operation that arrived late. This is done until the state reaches the current time. This state would be saved and displayed to the user. Concluding, timewarp would cause the train to be moved to the correct position on the correct branch of the tracks.

B. Correctness

In order to prove that timewarp ensures correctness, we show by complete induction that at any site all states that are saved in the list of states are the same that would have been calculated by the virtual perfect site P. This implies correctness if T is chosen such that it is smaller than the resolution of the physical clock and if the algorithm can be executed arbitrarily fast.

- Prerequisites: Let i be a site and t be a time for which $\forall w,\ t^* \leq t,\ o_{w,t^o,t^*} \in O \mid R_i(t, o_{w,t^o,t^*})$. Let i use timewarp.
- Claim: $\forall s_{i,n} \in L_i^s \mid s_{i,n} = s_{P,n}$
- Induction Base: By definition $s_{i,t_i^I} = s_{P,t_i^I}$
- Induction Hypothesis: We assume that the claim holds for all $n \leq n^*$
- Induction Step: Let $s_{i,m}$ be a state in L_i^s with $m = min\{k|(s_{i,k} \in L_i^s) \wedge k > n^*\}$. Then we know from Step 3 of the timewarp algorithm, that $s_{i,m}$ has been determined based on a state $s_{i,r}$ with $r \leq n^*$, where we know that $s_{i,r} = s_{p,r}$ because of the induction hypothesis. As described in Step 3 of the timewarp algorithm, from the time r on timewarp has calculated the state before each event according to the rules of the application. Also each event has then been executed based on that state, again according to the rules of the application. This is done until m is reached. Since this describes exactly the behavior of the virtual perfect site P it follows that $s_{i,m} = s_{p,m}$, concluding the induction step.

C. Computational Complexity

In order to determine whether timewarp provides correctness, we have assumed that the algorithm can be executed without consuming time. In a real-world system we can no longer make this assumption and must therefore take the computational complexity of timewarp into account.

In the following, we determine the computational complexity of timewarp depending on the number of participants (n) in a session. To do so the following assumptions are made:

- The number of operations scales linearly with the number of participants, i.e., each user produces a constant average number of operations per time interval, which seems to be realistic.
- The maximum amount of time an operation arrives late (i.e., the amount of time it arrived after t^*) is a constant determined by the network, and it is independent of the number of participants in the session. This requires that the participants of a session do not influence the maximum delay of operations in the network. It seems a safe assumption as long as the data transmitted for the session over the network is only a small part of the total traffic carried by the network.
- The amount of state information increases linearly with the number of participants. I.e., each participant contributes a certain amount of state to the overall state of the application. In a distributed virtual environment this could be the virtual representation of the user.
- The determination of a new state, based on a previous state without accounting for any operations, is $O(n^2)$. Typically the calculation of the new state requires that all parts of the state introduced by the individual participants are checked against each other. For example, detecting collisions between the virtual representations of the users would require checking all of their positions against each other. If

the parts of the state can be calculated independent of each other then the complexity would only be $O(n)$.

- The execution of one operation has a complexity of $O(n)$. This is based on the observation that operations often need to be checked against each part of the state that is introduced by the participants in the session. For example, firing a laser beam requires that each virtual representation of a user is checked whether it got hit.

With these assumptions it is clear that any algorithm that calculates the complete state of the application at each site will have at least a complexity of $O(n^2)$ for two reasons: 1) the calculation of a new state based on an old state is $O(n^2)$ and 2) a number of operations which linearly increases with n needs to be executed with $O(n)$ each.

Claim: The complexity of timewarp is $O(n^2)$ with n being the number of participants in a session.

Proof: Each cycle through all steps of timewarp is started once the previous cycle is finished. The number of cycles does not increase with the number of participants in the session. Therefore, it is sufficient to show that each cycle through all steps is $O(n^2)$. Each step of timewarp is executed sequentially. It is thus sufficient to show that each step has a computational complexity of at most $O(n^2)$.

- *Step 1.* The reception and storage of operations in a sorted list during a constant period of time can be done with a computational complexity of $O(log(n))$.
- *Step 2.* The determination of the starting state from a sorted list is independent of the number of users. The retrieval of that state is $O(n)$ since the state grows linearly with the number of users.
- *Step 3.* This step describes an iteration over all clock readings for which an operation exists that needs to be executed at that reading. In the worst case at each reading at least one operation will have to be executed. If we assume that the physical clocks tick in discrete steps, this iteration has therefore a constant number of steps depending only on the amount of time covered by the timewarp. Per our assumptions the maximum amount of time covered by the timewarp is independent of the number of participants. Therefore, the number of steps in this iteration is constant. With this knowledge it is sufficient to show that each step in the iteration has a computational complexity of at most $O(n^2)$. During each step a new state is calculated which is $O(n^2)$, and all operations that need to be executed at this time are executed. Since the number of operations per period of time increases linearly with the number or participants, this is also $O(n^2)$.
- *Step 4.* Calculating the final state is $O(n^2)$, and storing it can be done at $O(n)$ since the size of the state increases linearly with the number of users.

Since all individual steps require at most $O(n^2)$ and since the number of timewarps does not grow with n, timewarp has a complexity of $O(n^2)$.

An implementor of timewarp must be careful to make sure that timewarps are not executed in response to each operation that arrives late. This would result in an increase of the computational complexity by a factor n. Step 1 defined above prevents this and makes the number of timewarps independent of the number of participants. Furthermore, an implementor should ensure that in step 3 the state is calculated only once for each reading of the physical clock. If this is not taken into account and the state is determined for each operation, then the computational complexity would again increase by a factor n.

D. Further Considerations

In order to use timewarp in a real-world system, there are several issues that need to be addressed. First of all it is not possible to save all past operations and states for an unlimited time. Therefore, elements in the list of operations (L_i^o) and in the list of states (L_i^s) need to be deleted. This should be done so that it is very unlikely that a timewarp needs to be conducted to a time for which the operations or states have been deleted. In todays networks several minutes would capture most cases, even if the time required for loss detection, retransmission and congestion control is taken into account. If an operation arrives that precedes the operations and states in the lists, then some alternate means must be used to repair the state. This could include disconnecting the application instance and performing a late join on the session.

Another concern is how to conceal the visual artifacts that may result from short-term inconsistencies corrected by timewarp. Examples are the jumping train from the previous sections, or a player in a first person shooting game that gets revived. Generally this requires the usage of concealment strategies that are outside the scope of this work. One possible approach to minimize the occurrence of these artifacts is to delay the visual rendering of significant events such as a player's death in a networked game.

When timewarp is combined with local-lag, then the constant time T in step 1 of timewarp will additionally increase the average response time of operations and decrease the likeliness that a noticeable short-term inconsistency occurs. The reason for this is that the time interval T is used to collect all events that should be executed in this iteration of the timewarp algorithm. This rises the question whether the time interval T can be used to replace local-lag. This is not the case: local-lag is added individually to the execution time of each operation, allowing that each operation is delayed by the amount of local-lag before a short-term inconsistency will occur. The interval T on the other hand just determines the amount of time between iterations of the timewarp algorithm - the increase in response time and protection from short-term inconsistencies will depend on when the operation has been issued in relation to the starting time of the interval T and may thus be different for distinct operations. As a result, a combination of local-lag and timewarp does make sense: T should be chosen to match the framerate of the application while the determination of local-lag should take into account that the execution of an operation is additionally delayed by $\frac{T}{2}$ on average.

A similar approach as described here, which is also termed timewarp, has been used in the context of parallel discrete event simulation (PADS). A good introduction to PADS can be found in [14]. In PADS a simulation is

split into different parts, and each part is simulated on one computer. Each part may give input to or receive input from the other parts in form of operations. Timewarp in PADS allows the individual parts to be simulated independently. When an input arrives that should have been taken into account earlier on, a timewarp is performed to that time, and the simulation is recalculated. This may lead to the problem that operations sent earlier by the node that performed the timewarp are no longer valid. Therefore, it may be necessary to send anti-operations to those parts that the original operations were sent to. This in turn may cause further timewarps at those parts, triggering a cascade. Therefore, timewarp is controversial for PADS. However, in the context of consistency, there is no need for anti-operations, since each site maintains a full copy of the application's state. This is the main difference between applying timewarp to PADS and using it for maintaining consistency in continuous replicated applications.

VII. Experimental Results

In order to verify the theoretical observations and to show that local-lag as well as timewarp can be used in real applications, we have implemented a very simple networked computer game. In this game each player controls a spaceship which can accelerate, decelerate, turn, and shoot with a laser beam of a certain length. Each spaceship has three hit points: each time it is hit by a laser beam, one of the hit points is subtracted. If no hit points remain, the spaceship is removed from the game. The number of players is only limited by the processing power of the participant's computers. The game has been developed completely in Java under JDK1.3 using IP multicast for communication. A screenshot of the game can be seen in Figure 4.

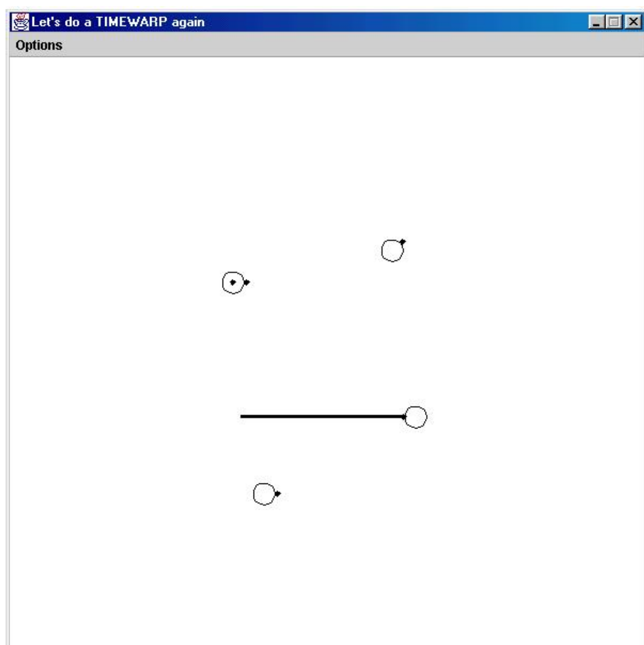

Fig. 4. Screenshot of the timewarp game

All experiments where conducted in a LAN where the network delay was below 1 ms. In order to simulate arbitrary network delays the game allows to delay incoming packets by a fixed amount of time. This amount is chosen at the startup of the application. The computers used for the experiments where off-the-shelf PCs with Athlon 600 processors. The total number of participants in the experiments was 19.

In a first series of experiments two users competed with each other. The simulated network delay was set to 0 ms, the amount of local-lag was increased stepwise in increments of 40 ms from 40 ms to 200 ms. The participants were asked after each game whether they had noticed any odd behavior in the game. The results where as follows: no user noticed local-lag of 120 ms and below. The average amount of local-lag that seemed to be noticeable was 160 ms.

In a second set of experiments we used no local-lag and increased the simulated network delay from 40 ms to 200 ms. This was done to see when the visual artifacts that occur when a state is repaired by timewarp are perceived by the user. No participants reported that they noticed odd behavior for values below 120 ms. On the average the visual artifacts were recognized at 140 ms.

In order to evaluate whether the combination of local-lag and timewarp performs superior to completely compensating the network delay with local-lag or not compensating it at all, we conducted a third series of experiments. Here the simulated network delay was increased from 80 ms to 400 ms in steps of 80 ms. In each experiment the amount of local lag was chosen to be half of the network latency. As a result, no user noticed odd behavior below 240 ms simulated network delay. On the average odd behavior was noticed at 290 ms delay.

While the actual values determined by these experiments are very likely to be specific for the application and possibly for the setup of the experiments, they do show that a broad range of network delays is tolerated by the users. Furthermore, the third set of experiments illustrated that the introduction of local-lag can significantly increase this range.

In another experiment we used the game to evaluate the computational complexity of timewarp. In order to do so, we replaced the network part of the game with code that simulated an arbitrary number of remote players. This code allows to provide a value for the maximal network delay that an operation encounters. For each operation the network delay is then drawn from a uniform distribution between 0 and the maximum value. Each simulated participant issued one arbitrary operation with a likeliness of 75% every 50 ms leading to an average of 15 operations per second per simulated participant. The waiting time in step one of the timewarp algorithm was selected to be 50 ms.

We increased the maximum network delay from 0 ms to 2000 ms in increments of 500 ms, and we increased the number of participants from 1 to 201 in increments of 50. For each combination we ran 500 iterations of the timewarp algorithm and measured its duration as well as the num-

ber of events that had to be executed. The results from measuring the number of operations per timewarp are depicted in Figure 5. These results confirm that the number of events increases linearly with the number of players as well as with the maximum amount of network delay.

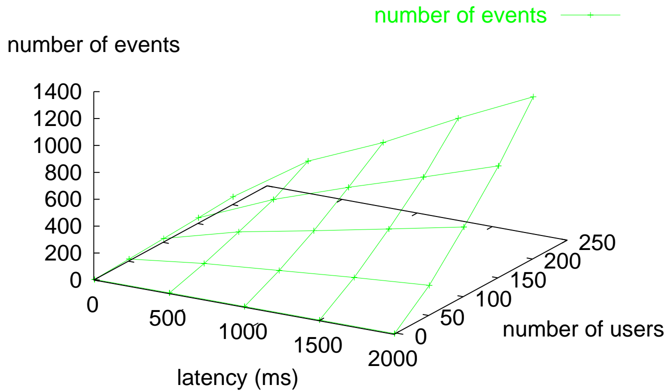

Fig. 5. Number of events per timewarp iteration

Figure 6 shows the amount of time required to complete all steps of the timewarp. As predicted by the theoretical analysis, the timewarp duration increases with $O(n^2)$. Also it can be seen that the amount of time required for a timewarp increases linearly with the maximum network delay. The reason for this is that linearly more events have to be executed but the size of the state (e.g., the number of spaceships) remains constant. Adding a participant linearly increases the number of events and the volume of the state.

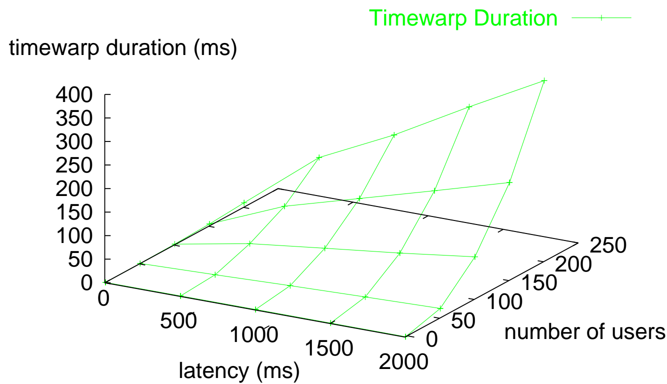

Fig. 6. Time required per timewarp iteration

The most noteworthy implication from Figure 6 is that timewarp requires only very limited computational resources if the number of participants is sufficiently small (e.g., below 50). It should be noted that these figures were reached using an unoptimized Java implementation. We expect that with current off-the-shelf PCs, at a frame rate of 20-25 frames per second, 100 to 150 participants could be supported with an optimized implementation using C or C++.

VIII. Dead-Reckoning

An approach that is commonly used to guarantee that the consistency criterion is satisfied in distributed virtual environments (e.g., multi-user virtual reality applications and battlefield simulations) is a combination of state prediction and state transmission. In this approach the application "knows" how the objects in the virtual environment behave over time when no user interaction occurs. Examples are a plane that will fly straight at constant speed or a projectile that will take a course dictated by the physical law of gravity. The prediction of the behavior of objects is called dead-reckoning.

Each object for which dead-reckoning is performed has a single controlling application instance, e.g., for a plane this will be the application instance of the pilot. Only the controlling application may issue operations that change the state of the object. The controlling application instance is responsible to inform peer application instances when the state of the object deviates by more than a certain threshold from the predicted state. In the event of a deviation, the controlling application transmits the complete state of the affected object to all peers. Upon receiving this information, an application discards the outdated state and uses the new state to perform dead-reckoning from here on.

User-initiated operations are applied to the local state of the affected object. The object will thus be put into a state that differs significantly from the predicted state, thereby requiring the controlling application to transmit the new state to its peer applications. Hence, operations are not transmitted explicitly but as part of an object's state. Furthermore, the states are transmitted unreliably. In order to repair packet loss in the network, the controller of an object transmits its state at regular intervals in the form of so called heart-beat messages.

One important property of dead-reckoning is that each application instance is only responsible for the objects it controls. For example, a collision between two objects that an application instance does not control is not discovered by that application instance. Instead the application instance relies on the application instances that do control these objects to transmit their state if a collision is detected. These events can be considered as artificial operations. The benefit of this approach is that the computational complexity required by dead-reckoning is reduced to $O(n)$ since it is not necessary to check for interaction between arbitrary objects, as is the case with timewarp. The drawback is that the number of events and thereby the number of short-term inconsistencies will increase.

In order to apply our consistency and correctness criteria to dead-reckoning, we say that a site j has received an operation o_{i,t^o,t^*} if it has received a state update that includes the effect of that operation. With this it becomes clear that the consistency criterion is fulfilled by dead-reckoning: if

at time t two sites i and j have received sufficient state updates so that they know of all operations (including the artificial ones) with $t^* \leq t$ then they will have the same state. This is true since in this case they use the same data for predicting the state of all objects.

While dead-reckoning provides consistency it cannot guarantee correctness. The reason for this is that dead-reckoning transmits state information and not information about events. Combined with the fact that state updates are transmitted unreliably it is not possible for an application instance that receives a state update for an object to determine how that state came to be. This in turn may lead the application instance to calculate an incorrect state for the objects it controls and distribute that state to the other application instances.

This problem is illustrated by the following example. Consider a car racing game where two cars A and B drive side by side. At some point in time the application controlling car A receives a state update for B that puts B on the other side of car A. If some of the state updates that precede this update for B were lost in the network, there is no way for the controlling application of A to determine how B got to this position. It is therefore not possible for the controlling application of A to determine whether the two cars collided, or whether B slowed down, moved over, and then sped up again. The virtual perfect site P as defined above will not have that problem since it receives all operations[1] in time and calculates its state based on this information. Therefore *dead-reckoning cannot guarantee correctness.*

It should be stressed that this problem is not restricted to situations where packet loss occurs. An incorrect state can also be reached if the controller of an object receives information about the interaction between two other objects sufficiently late and thus did not take this interaction into account when calculating its own state.

The tradeoff between short-term inconsistencies and the responsiveness of the application can be conducted with dead-reckoning in the same way it is done in combination with timewarp. In [4] the authors apply the principle of local-lag (or bucket synchronization as they call it) to a fully replicated game that also employs dead-reckoning. In this game the local state update is delayed by a fixed amount of 100 ms giving the state update time to reach the remote game instances.

Comparing the consistency-related characteristics of dead-reckoning and timewarp leads to the following results:

- dead-reckoning and timewarp provide consistency,
- only timewarp provides correctness,
- dead-reckoning has a computational complexity of $O(n)$ while the complexity of timewarp is $O(n^2)$,
- the reduced computational complexity of dead-reckoning leads to artificial operations which increase the potential for additional short-term inconsistencies.

The choice between dead-reckoning and timewarp as mechanisms to maintain consistency has further implications that are not necessarily consistency-related. Dead-reckoning requires that the *state* of each object be transmitted for each operation. This is only feasible if the state of the object is sufficiently small. Timewarp requires the reliable exchange of *operations*, therefore the state of objects may be arbitrarily large without causing problems. On the other hand, a reliable transport of the operations must be ensured for timewarp.

An interesting limitation of dead-reckoning is that only the controller may issue operations that change the state of an object. If two or more users want to interact simultaneously with the same object, it is thus required that the operations of those users whose applications are not the controllers of the object must transmit the operation to the controller. This potential detour of operations increases the network latency they encounter.

The discussion shows that both approaches have individual advantages and drawbacks. Neither of them seems to be superior for all imaginable applications.

An example where the dead-reckoning approach is used successfully, is the battlefield simulation protocol DIS (Distributed Interactive Simulations) [8]. In a typical battlefield simulation the state of the relevant objects is small (position and velocity) in size and relatively easy to predict. Each object has only one controller (e.g., the pilot of a plane, the driver of a car, etc.). The number of participants in a battle field simulation can be extremely high, on the order of up to 100000. Since the fate of individual objects is only of secondary concern, correctness is not required for this application class. Therefore dead-reckoning is a perfect fit for large-scale battlefield simulations.

On the other hand, for networked computer games with a limited number of participants, correctness may be vital to ensure fairness and to prevent cheating. Therefore, we believe that timewarp will have an edge for those applications. Similarly, CSCW applications with dynamic objects would profit from timewarp since the state of the objects may be to large to be transmitted frequently [6].

IX. Summary

In this paper we investigated mechanisms to provide consistency for replicated *continuous* applications. It was shown that the mechanisms from the discrete domain are not suitable for this application class since they ignore the need to execute operations at the correct point in time. In order to get a better understanding of the overall problem, definitions for the terms *consistency* and *correctness* were given and the tradeoff between short-term inconsistencies and response time was explained. We proposed the concept of local-lag, i.e., artificially delaying local operations to optimize the consistency related fidelity of the application. A timewarp algorithm was then introduced to guarantee consistency. Moreover, it was shown that timewarp also provides correctness. It has a computational complexity of $O(n^2)$, which is optimal for applications where each application instance maintains and calculates the complete state. In order to demonstrate the viability of our ideas, a simple networked computer game was implemented. Based

[1] artificial operations are ignored by P, since they can be derived from the other operations if the complete state is calculated as it is done by P

on this game the theoretical results were shown to hold true in a practical implementation. Finally, timewarp was compared to the well known dead-reckoning approach. We concluded that timewarp and dead-reckoning are suitable for different applications; none dominates the other in all aspects.

Acknowledgments

We would like to thank Mirko Friedrich, who implemented the main part of the timewarp game.

References

[1] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD Conference on Managements of Data*, Portland, OR, USA, 1989, pp. 399–407.

[2] 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 Interactions*, vol. 5, no. 1, pp. 63–108, March 1998.

[3] C. Sun and D. Chen, "Consistency maintenance in real-time collaborative graphics editing systems," *ACM Transactions on Computer-Human Interaction*, vol. 9, no. 1, pp. 1–41, March 2002.

[4] C. Diot and L. Gautier, "A distributed architecture for multiplayer interactive applications on the internet," *IEEE Network Magazine*, vol. 13, no. 4, July/August 1999.

[5] E. Frécon and M. Stenius, "DIVE: A Scalable network architecture for distributed virtual environments," *Distributed Systems Engineering Journal*, vol. 5, no. 3, pp. 91–100, 1998.

[6] M. Mauve, "TeCo3D - A 3D Telecollaboration Application Based on VRML and Java," in *Proc. of Multimedia Computing and Networking (MMCN'99 at SPIE'99)*, 1999, pp. 240–251.

[7] S. K. Singhal, *Effective Remote Modeling in Large Scale Distributed Simulation and Visualization Environments*, Ph.D. thesis, Department of computer science, Stanford University, 1996.

[8] S. Srinivasan, "Efficient Data Consistency in HLA/DIS++," in *Proc. of the 1996 Winter Simulation Conference*, 1996, pp. 946–951.

[9] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," in *Proc. of the ACM 1998 conference on Computer Supported Cooperative Work (CSCW'98)*, Seattle, Washinton, USA, 1998, pp. 59–68.

[10] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[11] S. Teal and A. Rudnicky, "A performance model of system delay and user strategy selection," in *Proc. of Human factors in computing systems 1992*, 1992, pp. 295–305.

[12] B. Schneiderman, "Response time and display rate in human performance with computers," *ACM Computing Surveys*, vol. 16, no. 3, 1984.

[13] S. Card, T. Moran, and A. Newell, *The psychology of human-computer interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[14] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.