Late Join Algorithms for Distributed Interactive Applications

Jürgen Vogel, Martin Mauve, Volker Hilt, Wolfgang Effelsberg
Praktische Informatik IV, University of Mannheim, Germany
{vogel, mauve, hilt, effelsberg}@informatik.uni-mannheim.de

Abstract— **Distributed interactive applications such as shared whiteboards and multi-player games often support dynamic groups where users may join and leave at any time. A participant joining an ongoing session has missed the data that has previously been exchanged by the other session members. It is therefore necessary to initialize the application instance of the late-comer with the current state. In this paper, we propose a late join algorithm for distributed interactive applications that provides such an initialization of applications. The algorithm is scalable, robust and can be easily adapted to the needs of different applications by means of late join policies. The behavior of the late join algorithm and the impact of design alternatives are investigated in detail by means of an extensive simulation study. This study also shows that an improper handling of the late join problem can cause very high application and network load.**

Index Terms— **Late Join, Distributed Interactive Applications, Consistency Control, RTP/I.**

I. Introduction

A *distributed interactive application* allows a group of users connected via a network to interact synchronously with a shared application state. Typical examples of distributed interactive applications are shared whiteboards, distributed virtual environments, and multiplayer computer games with network support. In order to achieve high responsiveness, scalability, and robustness, applications belonging to this class often rely on a replicated architecture and group communication. With this approach, a separate application instance is running on each participant's computer, and each instance maintains a local copy of (parts of) the shared application state.

One fundamental problem that these applications need to address is the support of participants who arrive late and wish to join an ongoing session: a latecomer needs to be initialized with an appropriate part of the current shared state before the user is able to participate in the session.

The late join problem is challenging in particular if the state of a distributed interactive application is large and complex. Most existing distributed interactive applications implement some form of support for latecomers without further investigation of alternatives and consequences. As we shall show, this may lead to high network and application loads as well as to consistency problems that could be prevented with an appropriate late join mechanism.

In this paper, we investigate the late join problem for distributed interactive applications in detail. We propose a generic solution that is scalable, robust, and flexible so that it can be employed by arbitrary applications belonging to this class. Scalability and robustness are reached by the usage of a distributed algorithm in combination with group communication. We show that the number of network groups used for late join purposes can significantly influence the scalability of the late join algorithm. In order to be flexible, we propose the use of a policy model that allows a late-joining application to structure and optimize the initialization process according to its specific needs.

The remainder of this paper is structured as follows. In Sect. 2, we examine the class of distributed interactive applications. In particular, a model is presented that allows an application-independent discussion of the late join problem. In Sect. 3, we briefly outline existing work and derive desirable characteristics of a late join mechanism. In Sect. 4, we propose our own late join algorithm, which has three different variants. All three variants are evaluated and compared by means of simulation in Sect. 5. The paper ends with a conclusion in Sect. 6.

II. Distributed interactive applications

In this section, we give a brief overview of the common characteristics of distributed interactive applications. On this basis, it is possible to discuss the late join problem and its solution in an application-independent way. A more detailed presentation can be found in [MHKE01].

A. States and events

A distributed interactive application has a state. For example, the state of a shared whiteboard is defined by the content of all pages present in the whiteboard. Each application instance maintains a local copy of at least parts of this state. For all instances these local copies should be reasonably similar. It is therefore necessary to synchronize the local copies of the application's state among all participants, so that the overall state is kept consistent.

The state of a distributed interactive application can change for two reasons – either by the passage of time or by events. The state of an application 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 the distributed application instances since each user's instance can independently calculate the required state updates. An example for such a state change is an animated 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 application, e.g., the user makes an annotation on a shared whiteboard page. Typically, information about events needs to be exchanged among all application instances in order to keep all state copies consistent. This can be done either by distributing the state of the application after executing the event or by announcing the event itself.

B. Partitioning the state into subcomponents

In order to provide for a flexible and scalable handling of state information, it is desirable to partition an application's state into several independent subcomponents. Examples of subcomponents are 3D objects (an avatar, a car) in a distributed virtual environment or the objects displayed on the pages of a shared whiteboard presentation. In addition to breaking down the complete state of an application into more manageable parts, such partitioning allows the participants of a session to track only the states of those subcomponents in which they are actually interested. We call subcomponents that are currently needed by an application instance to display the state to its local user *active*, whereas *passive* subcomponents are currently invisible, and their state cannot be changed by the user.

C. Consistency control

A replicated distribution architecture requires mechanisms that ensure the consistency of the shared application state. We expect that a distributed interactive application employs an appropriate mechanism such as operational transformation [SJZ+98], dead reckoning [SZ99], or timewarp [Mau00]. However, in order to support these mechanisms, it is important that the state of a subcomponent delivered by a late join algorithm be consistent (i.e., not missing event information), so that it can be used as a starting point for the consistency mechanism. A late join algorithm therefore needs to guarantee that a latecomer is supplied with a consistent initial state. Thereafter the consistency mechanism of the distributed application will take over.

D. RTP/I

In order to be able to specify and implement functionality that can be reused for all distributed interactive applications, it is useful to employ a standardized application-level protocol. RTP/I [MHKE01] is a protocol framework developed specifically for this purpose. It captures the common aspects of distributed interactive applications as described above. RTP/I consists of two parts: a data protocol for the transmission of states and events and a control protocol for the exchange of metadata about the participants of a session and about the subcomponents present in a session. The protocol elements encode sufficient semantics to develop generic services such as recording [HMKE99], consistency control [VM01], or the late join algorithm described here. All applications that base their communication model on RTP/I can integrate our late join service with minimal effort. Applications that do not rely on RTP/I can still use the algorithm described here by adapting its implementation to their specific communication model.

III. Existing approaches

Existing late join algorithms can be separated into approaches that are handled by the transport protocol and those that are realized at the application level. Application-level late join algorithms can be subdivided further into centralized algorithms and distributed algorithms.

Representatives of the first category are reliable transport services that offer late join functionality. An example is the Scalable Reliable Multicast Protocol (SRM) [FJL+97]. A reliable transport protocol can offer the late join service by using its loss recovery mechanism to supply the late-joining application instance with all data packets missed since the beginning of a session. The application then reconstructs the current state from these packets. An example of an application using this mechanism is the shared whiteboard MediaBoard [Tun98].

A main advantage of this approach is its robustness: as long as there is one application instance present with the required information, a latecomer will be able to join the session. Furthermore, the approach is application independent, and an existing implementation can therefore easily be used in different applications.

However, the usage of transport protocol functionality to solve the late join problem has several major drawbacks:

- It is inefficient since a large part of the transmitted information may no longer be relevant for the current state. For example, an image on a shared whiteboard page that has been deleted later in the session is not relevant anymore. In general, it is more efficient to transfer state information than to transmit all transport packets that have led to that state. When editing a text, for example, it makes sense to transmit the state of the text of the latecomer rather than all the packets that contain the description of characters that have been typed or deleted. This becomes even more important when the overhead for packet headers is taken into account.
- Either the application has to be able to reconstruct every packet that has ever been transmitted since the beginning of the session or the transport service needs to buffer the transmitted packets indefinitely. This is clearly not acceptable for a large number of applications.
- The state of certain applications may not be reconstructible from a simple replay of packets. The problem here is that for a given application (such as an networked action game) an event is only valid at a certain point in time. Thus in order to reconstruct the state of such an application from outdated packets, the application would have to jump back in time to the execution point of each packet before processing it. It is by no means guaranteed that an application would be able to perform this task.

Because of these problems we generally view the replay of packets as inappropriate for late join support. Instead, the current shared state should be explicitly queried by the latecomer. This leads us to existing late join approaches at the application level.

The distinct advantage of application-level approaches is the usage of application knowledge to optimize the late join process. Centralized late join approaches require that a single application instance exists that is able to act as the late join server for the entire shared state. A late-joining instance may contact the state server, which will in turn deliver the relevant state information. Examples of where a centralized state server is used for late join purposes are the Notification Service Transport Protocol (NSTP) [PDK96] and the Java-Enabled Telecollaboration System JETS [SdOG98].

The main advantage of a centralized solution at the application level is its simplicity with respect to the handling of consistency-related issues. At the same time, a single state server results in the typical disadvantages of all centralized solutions. Main problems are the existence of a single point of failure (lack of robustness) and the high application load for the server, which might become a bottleneck. In particular for distributed interactive applications that employ a replicated architecture, it seems to be inappropriate to introduce a centralized server just for late join purposes.

An alternative is to assign the role of a late join server for different subsets of subcomponents to different application instances. This is done by the collaborative virtual environment MASSIVE-3 [GPS00], where application instances are dynamically determined to be responsible for supplying latecomers with the states of a certain subset of subcomponents. This reduces the danger that a single late join server becomes a bottleneck. However, it does not increase robustness since a late join server may still fail or become overloaded if multiple late-joining participants are interested in the same subcomponents at the same time. For these reasons, we believe that a solution to the late join problem should not rely on dedicated late join servers at all.

Distributed late join approaches seek to avoid the problems of a centralized approach by involving multiple application instances in the late join process. Usually, many instances are able to take the role of a late join server for a given piece of the shared state. The failure of any single instance can be tolerated in these cases without preventing a latecomer from joining the session. One application that uses a distributed late join approach is the network text editor (NTE) [HC97]. If a text line is changed in NTE (e.g., a character is inserted), the complete state of the line is transmitted to all participants. Thus, latecomers are supplied with the active text areas very fast. The existence of all other lines is announced via periodic session messages. A latecomer may then request the missing lines. The request is served by the application instance responsible for the last state modification (or, should this fail, by any other instance that has the required information).

The main benefits of distributed late join approaches at the application level are robustness and scalability as well as the usage of application-level knowledge to avoid the drawbacks of the transport-level approaches. One problem, though, is the lack of reusability. For example, it would be quite difficult to tailor the late join functionality implemented for NTE to use with a virtual 3D world. In this paper, we propose a generic and reusable solution that supports distributed late join functionality at the application level. It thus combines the advantages of a generic solution such as proposed for late join algorithms at the transport layer with the benefits of using application-level semantics.

IV. Late join algorithm

The analysis of existing work leads to some interesting insights: to be efficient, a late join algorithm must make use of application-level knowledge. A replay of all transmitted packets is generally not an acceptable solution. In order to be robust and scalable, the late join algorithm should not contain any centralized elements. In particular, this holds true for distributed interactive applications that already employ a replicated architecture. Finally, the implementation of a late join algorithm should be flexible so that it can easily be used with different applications.

In the following, we denote a late-coming application instance as *late join client* and an instance that provides state information to a client as *late join server*. The roles of client and server are dynamic, and an application instance might be both late join client and server.

A distributed late join algorithm is usually composed of the following three steps:

- The first task for a late join client is to determine the point in time at which the state of a given subcomponent should be requested. If certain subcomponents are more important than others, the client should be able to prioritize the important ones. For example, consider a shared whiteboard where usually exactly one page is visible at a given point in time. The state of this page should be requested first by the late join client. Since such a decision is application-dependent, we use a policy-based approach that allows the application to control the late join service in this respect.
- In the second step, a late join server needs to be selected for the required state information. Since we do not rely on any centralized elements, a late join server must be determined from the group of all application instances each time a late join client requests the state of a subcomponent.
- Finally, the state information must be transmitted from the selected late join server to the clients needing the information.

For the realization of this distributed late join algorithm, the following four key problems have to be solved: selection

of application instances, use of late join policies, distribution of late join data, and, finally, consistency preservation.

A. Selection of application instances

One requirement for a late join algorithm is to operate in a fully distributed fashion without relying on dedicated late join servers. The state of a subcomponent can thus be delivered to a late join client by any session member that holds a (consistent) copy of that state. Since it is likely that more than one session member qualifies for being a late join server, there must be rules for the selection of which application instance should reply to a given request for state information. Similarly, at any given time multiple latecomers may wish to request the same state at the same time. This too should be coordinated so that request implosions are prevented.

Scalable selection of at least one session member from a potentially large group is a common problem in group communication and is known as scalable feedback [NB99]. A feedback algorithm must guarantee that at least one participant is selected. The number of participants selected should be low – ideally exactly one. Finally, the algorithm should be scalable in terms of group size and introduce only a marginal delay.

We propose using the exponential feedback raise algorithm [FW01] for the selection of application instances in late join situations: each session member that would like to send a state request or state information picks a random number $x \in [0;1)$. If $x < \frac{1}{N}$, where N is the group size, the application instance is selected and acts immediately. If $x \geq \frac{1}{N}$, an exponentially distributed back-off timer is set with running time $t = T(1 + \log_N x)$, where T is the maximum desired latency until at least one member must have been selected. If the back-off timer of a session member expires, the member has been selected and transmits the state or state request to the group. All application instances receiving this information cancel their own timers. As the analysis in [FW01] shows, the exponential feedback raise algorithm has a good behavior with respect to the expected number of selections and the expected selection latency. This is also verified in our own simulations (see Sect. 5).

B. Late join policies

Usually it is not necessary for an application instance to be initialized with the entire shared state at the time of the late join. A prerequisite for such a partial initialization is that the application's state be partitioned into subcomponents. For each subcomponent, a late join client can decide when the state for that subcomponent should be requested using its application-specific knowledge. The use of such policies ensures that the mechanisms a generic late join service implements can be easily adapted to the needs of individual applications. Existing late join approaches lack this ability.

In a shared whiteboard session, for example, a latecoming application instance typically needs all active subcomponents (i.e., subcomponents belonging to the currently displayed page) immediately in order to enable the participation of the local user in the ongoing session. In contrast, the state of all passive subcomponents (i.e., subcomponents belonging to currently invisible pages) are usually needed only when they become active again. Having this knowledge, an application could decide to request the state of active subcomponents only and postpone the requesting of passive pages to a later point in time. One major advantage of this approach is that only those parts of the state are transmitted at the time of joining that are actually required to participate in the current session. Thus, the amount of data transmitted is typically very small, which also minimizes the delay encountered by the late joiner before being able to actively participate in the session. Furthermore, a state initialization can be spread over a longer period of time, which also spreads network and application load.

In order to select different policies for individual subcomponents, an application must be able to learn about the namespace of subcomponents when joining a session. Besides information about which subcomponents exist, it also needs some application-level knowledge about each subcomponent, i.e., if a subcomponent is active or passive and which part of the application's state it represents. On the basis of this metadata, a late join client can decide about an appropriate policy. The control protocol of RTP/I provides such information by means of periodic subcomponent reports [MHKE01]. Once the presence of a subcomponent has been detected, the application can assign a late join policy to it.

For our late join service, we have defined a number of policies (also see [VMG+00]) for requesting subcomponents such as: no late join, immediate late join, event-triggered late join, and network-capacity-oriented late join. An application may change the late join policy for a given subcomponent at any time to account for changes in the importance of subcomponents or network load.

1. Late join policy: No late join
This (trivial) policy is chosen by the application instance to indicate that it is not interested in a certain subcomponent. In a distributed virtual environment, this policy could be used for subcomponents that the user will never be able to see. By choosing the *no late join* policy, the overall amount of state information that is required for the initialization of the late join client is reduced.

2. Late join policy: Immediate late join
An application instance may choose the *immediate late join* policy for those subcomponents that are currently needed to participate in the session. States of these subcomponents are then immediately requested. The likelihood that multiple latecomers will profit from a single transmission of the subcomponent's state is rather low with this policy.

3. Late join policy: Event-triggered late join
For a number of applications it might be reasonable to request subcomponents only if they are the target of an event. As illustrated above, a shared whiteboard might request pages with the immediate late join policy only if

they are currently active. Other subcomponents could be requested at the time they become the target of an "activate page" event. This request strategy is implemented in the *event-triggered late join* policy. Besides deferring the request of state information until it is actually needed, this policy also significantly increases the likelihood that multiple latecomers will benefit from a single state transmission (if multicast is used for data distribution) since all application instances will encounter the triggering event at the same time. Thus, all latecomers who join an ongoing session between two successive events for a subcomponent will benefit from a single state transmission.

4. Late join policy: Network-capacity-oriented late join
For subcomponents where the state is not immediately required, an application instance may choose the *network-capacity-oriented late join* policy. With this policy, the late join service monitors the incoming and outgoing network traffic of the instance. Only if the traffic falls below a threshold set by the application is the state of the subcomponent actually requested. This policy tries to defer the request for a state until the load of the network is low.

Other policies are conceivable and can be integrated easily into our late join service. Naturally, all late join policies that delay the transmission of state information increase the probability that the last participant that possesses those data leaves the session.

Most existing distributed interactive applications apply the immediate late join policy. MediaBoard, for example, additionally orders all requests according to their priority for the user [Tun98].

C. Distribution of late join data

After a late join request has been issued and an appropriate server selected, the initializing state needs to be transmitted to the late join client(s). It is desirable to distribute these data by means of group communication (i.e., application-level or IP multicast). Point-to-point connections are likely to introduce a bottleneck if a small number of potential late join servers serve a large number of late join clients. Also, depending on the late join policy, a single state transmission may initialize multiple late join clients (e.g., when employing the event-triggered late join policy). Thus, application and network resources are used more efficiently with group communication.

Variant 1: One network group
The easiest solution is to transmit late join data to the same group (the so-called *base group*) as the regular session data. All applications presented in Sect. 3 choose this approach. Its main benefit is its low complexity. But at the same time, all session members, including those that are not late join clients, will receive late join data. This may result in large amounts of data being delivered to application instances that do not need it. This is the case in particular if the ratio of late join clients to other session members is rather small.

Variant 2: Two network groups
Alternatively, a separate late join network group for the transmission of state information can be established. We call this the *client group*. All late join clients participate in the client group. State requests still have to be distributed via the base group in order to find a late join server. But in response, the selected servers send the requested state to the client group. Once a client does not expect to receive further late join information, it should leave the client group.

Preliminary simulations have shown that, depending on the distributed interactive application, the chosen late join policy model, and the user behavior, it is very unlikely that a late join client will ever complete the late join process for all subcomponents. For example, consider a shared whiteboard session where a set of slides is presented. Even if the lecturer jumps backward to an older slide every once in a while, it is unlikely that all slides will be presented more than once in a session. Therefore, a client should leave the client group when it has not requested or received any useful state information for a certain period of time. Should the client discover at a later point in time that it needs the state of a subcomponent (e.g., an older slide has been reactivated), it simply joins the client group again.

Restricting the receivers of late join information to application instances that probably need the data is expected to reduce both network and application load for the entire group. However, it introduces additional costs for the management of the client group.

Variant 3: Three network groups
Distributing late join state requests over the base group as described above has two drawbacks. First, requests are received and processed by each session member. This burdens both the network and the application instances. Second, if a large number of session members receive state requests, there is a higher probability that the selection process will pick more than one application instance as a late join server, even with the exponential feedback raise algorithm described above. Because a state transmission is costly for the application (since it must extract the current state of a subcomponent) and the network, a major goal for a late join algorithm is to minimize the number of duplicate server selections.

Distribution of state requests can be restricted by using a third network group: if a limited number of potential late join servers form an additional multicast group, state requests can be transmitted directly to these servers via this *server group*. Provided that the participants of the server group are chosen well, they can provide all latecomers with the desired data, while the majority of application instances are not involved in the server selection process. Because fewer application instances receive a state request, it is to be expected that network and application load due to state requests and duplicate states can be reduced.

This approach raises two questions. First, who should be a member of the server group? Second, what happens if a state request fails because no potential server for a requested subcomponent is present in the server group? The latter problem can be solved by a second request round: if no server can be found for a subcomponent in the server

group, the request is sent to the base group. While this introduces an extra initialization delay, it guarantees that state requests are eventually answered.

The first question is more complex, and there are a number of criteria that need to be considered for an algorithm that decides upon the membership in the late join server group: for each subcomponent there should be a server present in the late join server group to reduce the initialization delay, the server group should be as small as possible to minimize network traffic, the server group should be stable without many join and leave operations, and the algorithm should be fair in that the task of initializing clients is shared among all application instances.

We propose the use of the following adaptive mechanism for the selection of members in the server group: an application instance joins the server group when it is selected by a request in the base group as described above (i.e., because there was no appropriate late join server in the server group). An application instance leaves the server group if it has not answered any state requests for a certain amount of time t_l[1]. t_l is an adaptive timeout value and is calculated as follows:

$$t_l = T_m \left(\gamma \frac{S_p}{S} + (1 - \gamma) \frac{R_a}{R} \right)$$

T_m is the average group membership time. This value is provided by the application. S_p is the number of subcomponents an application instance can provide as a late join server. S_p is set in relation to the total number of subcomponents S present in the session. The intention of this is that application instances that can serve a large percentage of the subcomponents should stay longer in the late join group. R_a is the number of late join state requests that have actually been answered by a late join server. This number is set in relation to the number of requests R that could have been answered by this application. The lower this percentage, the less important is the presence of the application instance in the late join group. Finally, γ ($\gamma \in (0;1)$) is a weight that trades the importance of the number of present subcomponents against the number of answered requests.

In order to increase the stability of the late join group and to reduce the initialization delay, the application can also define a minimum group size for the late join server group. Servers are allowed to leave only as long as the current group size is higher than that minimum. Summarizing, this approach seeks to build a late join server group with a small number of "powerful" servers.

D. Consistency preservation

A late join is a special situation with regard to consistency maintenance. The key problem is that a latecomer may receive an initial state that is inconsistent because information about one or more events is missing. For example, this can happen if an event is still en route to the late join server. If the late join server transmits the initial state before the event arrives, its effect would not be included in the state. Once the event arrives at the late join server, the consistency mechanism of the application will make sure that the state at the late join server is corrected. However, the late join client receiving such an inconsistent initial state may have missed the event and has no means of detecting that the state is actually inconsistent. A late join algorithm therefore must provide a mechanism to discover this problem and guarantee a consistent initial state. After such a state has been delivered to the application, the regular consistency mechanism will take over, using this state as its starting point.

In order to solve this problem, we propose to include information about the most recent events in each initial state transmitted by a late join server. This information is a standard part of the RTP/I protocol. Furthermore, periodic session messages are distributed to the base group. These contain information about the events included in the current state of each subcomponent. Each latecomer checks the state it has received against these session messages. If the check indicates that the latecomer has received an inconsistent state, then that state is discarded and requested again. If necessary, this will be repeated until the check is successful. A more detailed discussion of this iterative state request mechanism can be found in [VM01].

V. Simulation analysis

We have implemented a fully operational generic late join service that is now part of the RTP/I code library. The implementation contains all three late join variants discussed in Sect. IV that use either one, two, or three network groups for the communication between late join client and server. Furthermore, it integrates the flexible late join policy model.

Based on this implementation we have developed a simulation toolkit that was used to generate the results discussed in this section. This simulation toolkit enables us to analyze late join algorithms with respect to different design criteria (initialization delay, network and application load, etc.). The simulation emulates typical sessions of different kinds of applications. In the following, we present the results of simulating a shared whiteboard scenario and an online game scenario. The simulation results were determined by running the simulation 20 times for each scenario and discarding the three highest and lowest values before calculating the average.

A. Shared whiteboard scenario

For our simulation, we set up a typical shared whiteboard session where a presentation is given to a medium-sized group with a maximum of 100 participants. Periodically, each application instance generates an action from the set {join session, leave session, create new subcomponent, send event, activate or deactivate a subcomponent, do nothing} with predefined probabilities. The timespan between two actions of the same application instance is randomized between 0.8 and 1 s. The network delay among end systems is

[1] Note that an application instance also leaves the server group when the user leaves the session.

varied between 20 and 150 ms, and a loss-free transmission of data is assumed[2].

Typically, a shared whiteboard group is relatively stable during the presentation, with only a few members joining or leaving. Therefore, the session starts with 80 members, which is also the minimum group size throughout the session (i.e., an instance is allowed to leave only if there are more than 80 session members remaining). After initiation, there is a dynamic phase while additional participants join with a high probability; toward the end members leave more frequently. This behavior was reflected by exponentially decreasing the join probability (starting with 0.02) and by exponentially increasing the leave probability (ending with 0.1).

In our model of a shared whiteboard, the application's state is structured hierarchically, with each slide containing a set of graphical objects. Each object is represented by a subcomponent of its own. At a certain point in time the same slide is visible to all users. Thus, only a small subset of subcomponents is active at any given time. This model is based on long-term practical experience gained with the mlb, the multimedia lecture board developed by our group at the University of Mannheim [Vog01].

A good strategy for a late join client in a shared whiteboard session is to request the state of the current slide (i.e., all active subcomponents) by using the immediate late join policy. No other subcomponents are requested unless they become active (event-triggered late join). Notification about the subcomponent space and the presence of other participants is performed by the periodic session messages of RTP/I.

When giving a presentation with slides prepared in advance, user activity changing the application's state is relatively low. We chose the probability of creating a new subcomponent (e.g., to add a new annotation) as 0.01, the probability of generating an event (e.g., extending or moving an existing annotation) as 0.15, and the probability of moving to another active page as 0.05. During the simulated time of 10 min, an average of 550 subcomponents were created and 8,200 events were issued. This scenario leads to an average of 3,900 late join requests.

Fig. 1 shows the distribution of the average initialization delay generated by the three variants of our algorithm. The initialization delay for a late join client is measured as the time span between sending the first request and receiving the requested state (the time needed to extract a subcomponent's state and to transmit the data are equal for all variants and were not considered).

Even though the second variant employs an additional group for the transmission of states, its request-response mechanism is basically the same as in the variant with only a single group. Thus, it was expected that both variants would have similar distributions of the initialization delay with an average value of 386 ms for variant 1 and 371 ms for variant 2. The third variant with three groups causes a higher initialization delay, with an average of 432 ms, because it is possible that no appropriate server can be found in the server group and an additional request is necessary. After the (initially empty) server group had been formed, only in less than 20% of all state requests was a second request round necessary. The initialization delay is acceptable for all variants, even if the times for state extraction and transmission are not included in these figures.

[2]If an application does not use a reliable transport protocol, the late join algorithm can repair packet loss by repeating the server selection process. This will increase the initialization delay.

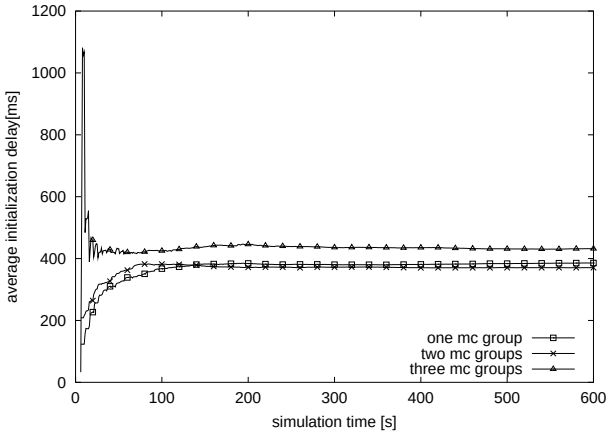

Fig. 1. Initialization delay

The composition of the groups for late join clients and servers is depicted in Fig. 2. It can be seen that late join activity in the shared whiteboard scenario is high at the beginning of a session. At first, many participants join within a short period of time, and late join requests cause members to join the server group as well. Since in this early phase of the session there are only a few subcomponents and all of them are active, the first wave of clients soon has received the complete application's state and leaves the client group. Later on, sporadic late joins occur so that the size of the client group increases slowly. Late joins and switching of active subcomponents (which may trigger applications to join the client group in this scenario) on the one hand and client timeouts as well as clients leaving the session on the other hand balance one another, so that the client group is relatively stable. The size of the server group fluctuates around the predefined minimum of ten members, which indicates that most requests can be answered by the current group members and additional servers are not necessary.

The network load caused by a late join algorithm can be measured by the cumulated numbers of states and state requests received by all application instances due to late joins. Since states might be large and their handling costly in terms of processing power, special care must be taken to minimize the number of transmitted and received states. Fig. 3 shows the simulation results for the cumulated number of states received by all application instances. When comparing the graphs of variants two and three on the one hand (7,800 and 7,700 states) with the graph of the first variant on the other (326,000 states), the dramatic effect of transmitting states over an extra late join client group becomes visible: a reduction by factor of 40.

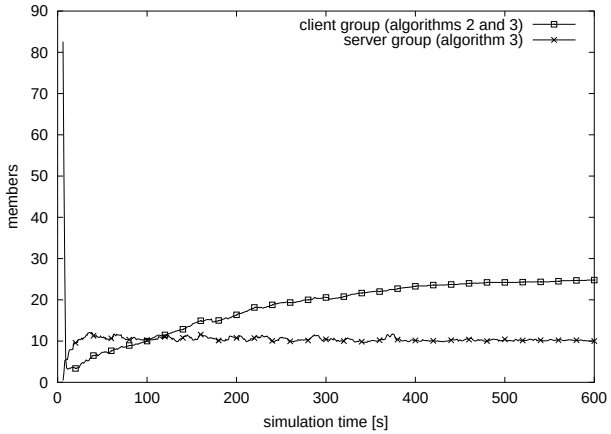

Fig. 2. Composition of late join client and late join server group

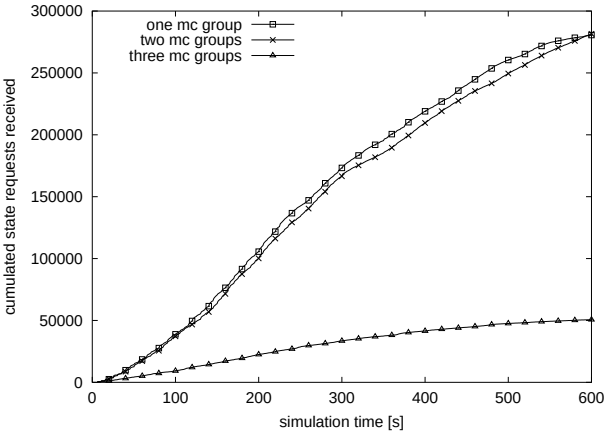

Fig. 4. Distribution of received state requests

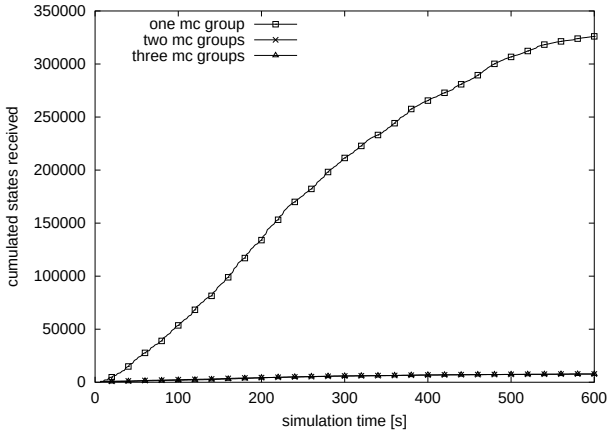

Fig. 3. Distribution of received states

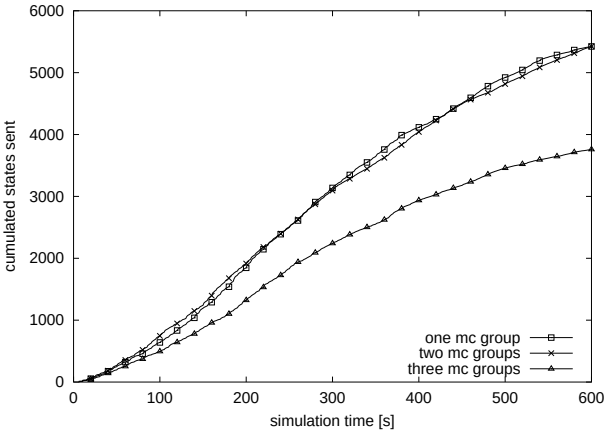

Fig. 5. Distribution of sent states

The second part of the network load is caused by the state requests exchanged (Fig. 4). Since both variants 1 and 2 transfer state requests via the application network group, similar graphs were expected. Distribution of state requests via the server group by the third variant results in a significantly lower number: in total all application instances received roughly 230,000 requests less than with variants 1 or 2. Even though requests are typically rather small (e.g., 24 bytes for RTP/I), their high number justifies setting up a server group.

The application load induced by a late join algorithm is mostly caused by four tasks. (1) Process received state requests (i.e., execute the selection of an application instance): As already discussed, the third variant results in by far the smallest number of received state requests and therefore in the smallest application load due to request processing. (2) Extract and send the state of a subcomponent if selected as late join server: Fig. 5 depicts the cumulated number of states sent for each variant. Since the third variant transmits state requests over the server group that is much smaller than the application group, the probability of selecting multiple servers is lower. Thus, the total number of 3,760 states sent in reply is 1,660 states lower than with variants 1 and 2. (3) Discard unneeded packets: As shown above, with variants 2 and 3 318,000 less states are received than with variant 1. (4) Manage additional multicast groups: The application load caused by the management of multicast groups can be approximated by the total number of join and leave operations. The application group has to be managed by all variants and is disregarded. During the simulation time, there were a total of 770 joins and 740 leaves for the client group and 1,000 joins and 990 leaves for the server group. The overall application load depends considerably on a specific application. But when considering the significant reduction of received late join traffic, it can be expected that the additional cost for group management will be negligible and that the third variant will result in the smallest load.

To summarize, using separate multicast groups for the transmission of late join requests and the distribution of late join states results in a significant reduction of total network load. Depending on the processing costs for the handling of state requests and states, these groups also produce a smaller total application load despite the overhead for group management. At the same time, however, there is a slight increase in the initialization delay for late join clients when a separate server group is used.

B. Online gaming scenario

In the second scenario, a multiplayer game with a maximum of 150 participants was simulated. The main characteristic here is that the composition of a session is much more dynamic than in the shared whiteboard scenario, with a continuous late join activity and a large variance in the number of session members. We had a minimum of 75 participants while the join probability of 0.2 and the leave probability of 0.01 remained constant throughout the simulation. The delay for the transport of data among application instances was chosen between 50 and 150 ms.

In this model of an online game, each participant has an individual view of the application's state with a set of active subcomponents that may be different from the set of active subcomponents of other participants. Thus, an application instance can also receive events for passive subcomponents. A late join client requests all subcomponents that are active for himself by the immediate late join policy and all other subcomponents by the event-triggered policy.

The probabilities for creating a new subcomponent and for generating events were set to 0.01 and 0.3, respectively. During the simulated time of 200 s, a total of 320 subcomponents and 9,300 events were issued, and approximately 23,000 requests for initialization with a certain subcomponent's state were sent.

As in the first scenario, the average initialization delay for the late join client was similar for the first and the second variants: 283 ms and 267 ms, respectively. For approximately 50% of all state requests, the third variant did not find an appropriate server in the server group, making a second request round necessary. As a consequence, the average initialization delay for the third variant was 420 ms. The rate of successful requests within the server group was lower than in the first scenario because each participant was interested in an individual set of subcomponents, and because there was a high rate of joining and leaving application instances.

The latter can also be seen in Fig. 6, which shows the composition of the client and the server group. In contrast to the whiteboard scenario, a high percentage of application instances are members of the client group, and the server group is rather unstable, with many servers providing only a few subcomponents.

The total network load due to late join activity was measured by the cumulated numbers of received states and requests. Since the first variant distributes all late join data via the application group, it produced a high network load with 741,000 received states and 1,343,000 received state requests. Variants 2 and 3 transmit states over a client group; they reduced the number of received states to 162,000 and 148,000, respectively. These savings are less than in the shared whiteboard scenario because of the large number of members in the client group. The server group of the third variant reduced the number of received requests to 600,000. This reduction is also lower when compared to the first scenario since the size of the server group was larger and more requests failed in the first round.

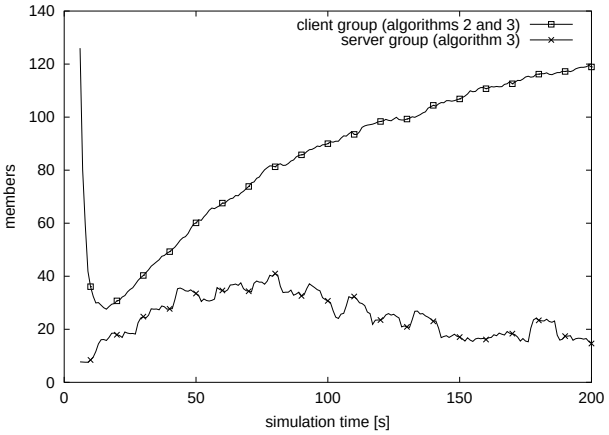

Fig. 6. Composition of late join client and late join server group

Concerning the application load, variants 1 and 2 both lead to a total number of 13,400 state transmissions, whereas 8,950 states were transmitted with the third variant. The management of the client group had to handle 1,690 join and 1,570 leave operations in variants 2 and 3. The dynamics of the server group was evident with 8,450 join and 8,430 leave operations. We conclude that depending on the application type, it can be expected that the third variant produces the lowest overall load, even though the proportion of the group management costs is higher than in the whiteboard scenario.

To sum up, introducing additional network groups for late join clients and servers saves a significant amount of application and network load. These savings are lower for applications with higher join and leave rates of session members as in the gaming scenario. The higher initialization delay for late join clients might be crucial for real-time applications. Variant 2, with a separate client group but without an additional server group, could therefore be a good fit for those applications.

VI. Conclusion

In this paper, we presented an algorithm for the initialization of latecomers in distributed interactive applications. A fully replicated approach and group communication were used in order to reach scalability and robustness. The algorithm was realized as a reusable service by employing a generic model for distributed interactive applications in combination with flexible late join policies. However, the basic concepts presented here can also be used as a basis for application-specific solutions to the late join problem.

By simulating two different scenarios for distributed interactive applications, we have shown that a carefully designed late join algorithm significantly reduces application and network load. Furthermore, the simulation provided insights into how to best distribute late join data to clients. We demonstrated that applications with a stable group membership such as shared whiteboards will benefit considerably from two additional network groups: one for the transmission of state information to late join clients and one for the transmission of state requests to the potential

servers. In contrast, very dynamic and time-critical applications (i.e., networked computer games) are likely to be best served by using only one additional network group for the transmission of states to the clients.

We have integrated our late join algorithm into two existing applications. TeCo3D [Mau99] is a 3D telecollaboration tool that allows a group of users to share VRML animations. For TeCo3D, we chose to use only one additional network group in order to minimize the initialization delay. The second application is the shared whiteboard mlb (multimedia lecture board [Vog01]). Here we opted for the variant of our algorithm that uses a client and a server group. Both applications use the policy model to request active subcomponents at once and all other subcomponents when receiving events.

Acknowledgments

The authors would like to thank Marcel Busse for implementing the late join simulation.

References

[FJL+97] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Leight-Weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):784 – 803, 1997.

[FW01] T. Fuhrmann and J. Widmer. On the Scaling of Feedback Algorithms for Very Large Multicast Groups. *Special Issue of Computer Communications on Integrating Multicast into the Internet*, 24(5-6):539–547, 2001.

[GPS00] C. Greenhalgh, J. Purbrick, and D. Snowdon. Inside MASSIVE-3: Flexible Support for Data Consistency and World Structuring. In *Proc. of ACM CVE 2000, San Francisco, USA*, pages 119–127, 2000.

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

[HMKE99] V. Hilt, M. Mauve, C. Kuhmuench, and W. Effelsbeg. A Generic Scheme for the Recording of Interactive Media Streams. In *Proc. of IDMS, Toulouse, France*, pages 291–304, 1999.

[Mau99] M. Mauve. TeCo3D: a 3D telecooperation based on VRML and Java. In *Proc. of SPIE Multimedia Computing and Networking (MMCN) 1999*, pages 240–251, 1999.

[Mau00] M. Mauve. How to Keep a Dead Man from Shooting. In *Proc. of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS)*, pages 199–204, 2000.

[MHKE01] M. Mauve, V. Hilt, C. Kuhmuench, and W. Effelsberg. RTP/I - Toward a Common Application-Level Protocol for Distributed Interactive Media. *IEEE Transactions on Multimedia*, 3(1):152–161, 2001.

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

[PDK96] J.F. Patterson, M. Day, and J. Kucan. Notification Servers for Synchronous Groupware. In *Proc. of ACM CSCW, Cambridge MA, USA*, pages 122–129, 1996.

[SdOG98] S. Shirmohammadi, J.C. de Oliveira, and N.D. Georganas. Applet-Based Multimedia Telecollaboration: A Network-Centric Approach. *IEEE Multimedia Magazine*, 5(2):64–73, 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.

[SZ99] S. Singhal and M. Zyda. *Networked Virtual Environments Design and Implementation*. ACM press, 1999.

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

[VM01] J. Vogel and M. Mauve. Consistency Control for Distributed Interactive Media. In *Proc. of ACM Multimedia 2001, Ottawa, Canada*, pages 221–230, 2001.

[VMG+00] J. Vogel, M. Mauve, W. Geyer, V. Hilt, and C. Kuhmuench. A Generic Late Join Service for Distributed Interactive Media. In *Proc. of ACM Multimedia 2000, Los Angeles, USA*, pages 259–268, 2000.

[Vog01] J. Vogel. multimedia lecture board (mlb) homepage. URL: http://www.www.informatik.uni-mannheim.de/informatik/pi4/projects/ANETTE/anetteProject2.html, 2001.