An Efficient and Flexible Late Join Algorithm for Interactive Shared Whiteboards

Werner Geyer, Jürgen Vogel, Martin Mauve
Lehrstuhl Praktische Informatik IV
University of Mannheim,
L 15, 16
D-68131 Mannheim, Germany
Tel: +49-621-1812615
Fax: +49-621-1812600
{geyer, jvogel, mauve}@pi4.informatik.uni-mannheim.de

Abstract. In this paper we propose a novel late join algorithm for distributed applications with a fully replicated architecture (e.g. shared whiteboards). The term 'late join algorithm' is used to denote a mechanism that allows a late-coming participant to join an ongoing session. Generally, this requires that participants in the session inform the latecomer about the current state of the shared application, so that he or she is able to actively participate in the session once the late join has been completed. While many existing distributed applications implement some sort of late join, a thorough discussion of late join algorithms has not yet been conducted. We therefore identify the key issues of late join algorithms and propose a set of requirements which a 'good' late join approach should satisfy. Based on these requirements, we evaluate existing late join algorithms and explain why we opted instead to develop a new, advanced late join algorithm for our own shared whiteboard. This late join approach is discussed in detail, and we show how it enhances existing approaches significantly while still being general enough to be used for arbitrary distributed applications.

1 Introduction

The availability of IP multicast in the Internet provides an excellent basis for the development of CSCW applications since multicast is a very efficient way to realize group communication. Currently the most popular multicast application scenario is video conferencing. While early research focused on the development of tools for the transmission of audio and video streams, in recent years, new multimedia applications in the field of telecooperation have emerged, e.g. shared whiteboards, shared editors, or distributed virtual reality applications. In contrast to the well-known application sharing approach, these newer applications are based on a replicated distribution architecture, i.e. a *collaboration-aware* instance of the application is running on each participant's machine. At any point in time such an application has a well-defined *state* which can be changed by *events* transmitted over the network, e.g. initiated by user actions. The obvious benefits of this approach are its high scalability, low responsiveness, and robustness against failures of single components.

However, a replicated architecture also implies a replication of the application data. Hence, a major drawback of this approach is that it requires complex mechanisms to ensure the consistency of the distributed data. While there are many approaches maintaining the consistency of a distributed application (e.g. total ordering of data, time stamps), the problem of efficient state initialization is still unsolved. This problem arises whenever a new participant joins an ongoing session. We call this *late join*. In order to enable the participant to follow the session, his or her instance of the application needs to be initialized to the current application state. The mechanisms to transmit the current application state to new members of a session are called *late join algorithms*. A state initialization of a replicated application by means of a late join algorithm is not only required by latecomers but also for recovery purposes in case of hardware or software failures.

In this paper we present a novel late join algorithm which has been developed in the context of a shared whiteboard known as the digital lecture board (dlb) [5] [3] [4]. The proposed mechanisms are general enough to be applicable to arbitrary replicated applications that require an efficient state initialization. Our approach keeps the initialization delay low for the latecomer while reducing the network and application loads. The late join algorithm has been successfully implemented and tested in the digital lecture board.

In the following Section we analyze key issues of the late join problem and identify a set of general requirements for late join algorithms. Section three covers existing work in the area of late join algorithms. In the main part of this paper we present a stepwise solution to the late join problem: our first approach is based on unicast network connections; the second and final algorithm makes use of the basic ideas of the first scheme but additionally relies on multicast to further improve performance. Finally, we discuss the generalization of our late join algorithm. Section five concludes the paper with a summary and an outlook.

2 Key Issues of Late Join Algorithms

The prime requirement for any late join algorithm is its ability to enable latecomers to participate in an ongoing session. This is typically achieved by getting the state of the distributed application from the current participants and initializing the application of the latecomer with this information. In the following we will use the term *late join client* to denote a new participant who requests a state initialization; the participant who is responsible for the retransmission of the current state at a certain point in time is called the *late join server*. The role of the late join server may be assigned dynamically.

From the latecomer's point of view the late join process can be divided into three distinct tasks: (1) identify what information is needed, (2) get the required information, (3) initialize the application so that it is consistent with the other participants' applications. In the following subsections we will discuss the key problems that have to be solved for different late join scenarios and will then propose a set of requirements which a 'good' late join algorithm should fulfill.

2.1 Late Join Problems

In a scenario with a dedicated late join server all three tasks can be handled in a fairly straight forward fashion. The latecomer asks the server about the currently required information, then proceeds to request this information and finally initializes the state of its application. The only challenge in this scenario is to make sure that state changes which happen during the late join are handled in a way that ensures a consistent state of the latecomer's application.

In a fully distributed environment with true peer-to-peer multicast communication each of the three tasks poses its own challenge. For the first task it is essential to get a consistent view on what information is needed to initialize a latecomer's application. Challenging problems in this area include knowing where and how this information is kept, as well as who is responsible for providing this information to the latecomer.

Because of the lack of a dedicated server, one main issue in task two is to identify who will reply to the information requests of the latecomer. A naive approach would let all current participants reply to the requests. However, for a group of size n this would lead to n-1 duplicate transmissions of this information. Since it is likely that more than one user will require a late join within a short period of time, the number of duplicates is likely to rise to $i(n-1)$ where i is the number of latecomers during a certain (short) time interval. In the worst case for $i=n/2$, the number of duplicates rises to $n^2/4$. This is clearly unacceptable; a late join algorithm thus needs to find a way to eliminate or at least to reduce the number of duplicate transmissions.

Finally, the initialization of the application is challenging because user interactions might have modified certain parts of the state while the late join was in progress. A late join algorithm in a peer-to-peer environment therefore needs to monitor the regular communication during the late join and adjust the state of the application to compensate for any user actions.

2.2 Late Join Requirements

Derived from the problems mentioned above, as well as from general design aspects of distributed applications, the following requirements for late join algorithms can be identified:

- **Consistency**. A late join should lead to a consistent state of the latecomer's application. With this state he or she should be able to actively participate in the ongoing session.
- **Robustness**. The failure of individual components such as a participant's computer or application should not prevent a future late join. Moreover, a late join is likely to be executed after an individual component (e.g. the application) has failed, in order to let the affected user (re)join the session.
- **Low network load**. The efficiency of a late join algorithm hinges on its sending only a minimal number of duplicates. In addition only the data which is really needed should be transmitted. For example, it might be efficient for a shared whiteboard to request only the state of the currently visible pages[1] rather than requesting the state of all pages ever shown in the session.

---

[1] Assuming that there exists some mechanism that recovers the state of existing pages which become visible at a later point in time.

- **Low application load**. Those participants who are not latecomers should not suffer too much from the additional data transmissions required for the late join.
- **Low initialization delay**. The late join should complete within a limited time interval which is acceptable to the user.

In the following section we will discuss existing late join approaches focusing especially on how they solve or fail to solve the problems mentioned above, and how well they meet the requirements for a 'good' late join algorithm.

3 Related Work

Existing late join algorithms can be separated into approaches which are handled by the transport layer and those which are completely realized at the application level. Application level late join algorithms can further be subdivided into client-server oriented algorithms and peer-to-peer approaches.

Representative of the transport layer category are reliable multicast transport services that offer late join functionality. A reliable multicast protocol can offer this service by using its loss recovery mechanisms to supply the late-joining application with all data packets missed since the beginning of the session. The application then needs to reconstruct the current state from these packets. An example of reliable multicast protocols which support this operation is the Scalable Multicast Protocol (SMP) [7].

While this approach is very simple and straightforward, it has three main drawbacks: first, protocol instances need to cache all data packets sent since the beginning of the session, resulting in high storage requirements at the transport level. Second, both application and network load are most likely much higher than necessary. Replaying all packets is likely to convey information that is no longer relevant, e.g. deleted pages in a shared whiteboard. Third, the initialization delay in this approach is rather high since the application is fully operational only after all packets have been received. The transport protocol cannot send those packets which are needed to display the currently active pages before transmitting other packets because it lacks all knowledge about packet contents.

The distinct advantage of application level approaches is the use of application knowledge to reduce initialization delay as well as network and application load. Client-server based late join approaches typically use a centralized data storage model in which the dedicated data server also acts as late join server. Late-joining clients usually connect to this server via unicast and request the part of the current application state that interests the client. Consistency is easily achieved since state changing events are only executed at the server and can simply be forwarded to the late-join client.

An example for the client-server approach is the groupware toolkit Clock [6]. Distributed applications implemented with Clock are based on the Model-View-Controller (MVC) architecture that separates the application data and its data manipulating methods (model), the user interface (view) and the event handling (controller). Each client has its own instance of the controller and the view whereas only the data server possesses an instance of the model. When joining a session, the view of the client requests all relevant data.

Centralized approaches have the advantage that the ability to perform a late join is inherent to their architecture, i.e. a late join basically equals a regular state update. However, centralized architectures also have the following well-known drawbacks: single-point-of-failure, as well as a high network load and a high application load for the server.

In the case of distributed applications with a peer-to-peer architecture, every participant in a session possesses a complete copy of the current application state. Therefore every participant is able to act as a late join server, and every late join client can have one or more different servers. The MediaBoard (mb) [9] is a representative of this category. Like its predecessor - the MBone whiteboard wb - the MediaBoard integrates the reliable multicast transport protocol within the application, following the Application Level Framing (ALF) paradigm [1]. The late join algorithm used by the MediaBoard is based on SRM [2], using its loss recovery mechanism to request the state necessary for the initialization. Since the protocol is located within the application, it can use priorities to realize a selective retransmission of the application state. The late join client first requests all the data needed to display the active page; data of all other pages are requested with a low priority.

The advantages of this approach are high robustness, a low network load and a low initialization delay. While it represents the state-of-the-art for late join algorithms, this approach still has some major drawbacks. First, all participants of a session will see the retransmission of the whiteboard pages, even though non-latecomers are not interested in this information. Second, the tight coupling between application functionality and loss recovery mechanism implies that a late join can only be realized by replaying the entire history of network events. A replay is inefficient since the current application state cannot be transmitted directly but can only be reconstructed through the sequence of events. In addition to the time-consuming reconstruction of objects (e.g. restoring text objects letter by letter), this also entails information being transmitted that is no longer relevant (e.g. delete events or consecutive

move events). Third, the granularity of the late join algorithm is limited to the original data packets, i.e. each packet has to be requested (and repaired) separately. A single collective request (e.g. to restore a complete page of the whiteboard) would be more efficient. Moreover it is not guaranteed that the replay of ‘old’ packets allow an application to restore the correct state. If the medium is time-dependent, this may not be possible.

4 A Selective Late Join Approach

The late join algorithms implemented in the digital lecture board belong to the category of application level solutions for a completely replicated architecture. Transport level solutions were not a viable alternative due to their restrictions mentioned previously. We also did not consider centralized client-server approaches since the digital lecture board - like most whiteboards - adheres to the paradigm of complete replication in order to avoid the well-known problems of centralized components.

In contrast to applications that comprise a reliable transport protocol (e.g. wb, mb) following the ALF concept, the dlb uses a reliable transport service below the application. While the ALF approach might generally be more efficient, we considered a clean cut between application layer and transport protocol functionality more important than a slight increase in performance. The choice is application dependent: for the dlb we did not expect a high performance gain by using ALF while it would have made the design of the software significantly more complex. While the separation of reliability mechanisms and application prohibits us from (ab)using the loss recovery algorithms for late join purposes, it also enables us to provide a very generic and thus ’clean’ solution to the late join problem. Moreover, separating reliability issues from the application facilitates application development since the developer can then concentrate on key design problems.

The discussion in Section 3 on related work demonstrated that a *low initialization delay* during the late join can only be achieved by reducing the amount of data transmitted to the minimum needed by the late join client to follow an ongoing session. With respect to whiteboards the selection of this minimal required data is rather straightforward. A shared whiteboard document consists of an arbitrary number of pages that contain an arbitrary number of graphical objects. At each point in time there is typically only one active page that is visible to all participants of a session. The data needed by a latecomer to follow an ongoing whiteboard session is a list of all available pages and the objects of the current (active) page. The objects of the remaining pages are requested only on demand, i.e. as soon as an arbitrary participant (including the late join client) activates such a page.

According to this selection scheme, we have developed two late join algorithms: a selective late join based on unicast transmission and its successor, a selective late join based on multicast transmission. Besides providing a low initialization delay by requesting only the necessary information, both algorithms are independent of the transport protocol and focus on the reduction of network and application load.

4.1 Selective Late Join based on Unicast Transmission

The first algorithm involves two fundamental steps: in the first, a late join server is selected from the multicast group and a unicast connection is established between the late join client and the late join server. In the second step, the required data is transferred.

Step 1: Selecting the Late Join Server. To select a particular participant from the multicast group, we realized an anycast mechanism on the level of the transport protocol. This mechanism is basically very similar to the repair mechanism of SRM [2]. Notice that the anycast mechanism could just as well be realized within the application in order to maintain the independence of the transport protocol. The time-sequence diagram of an anycast request is shown in Figure 1.

In this example client C sends its IP address and port number plus information identifying the requested data with a special Application Data Unit (ADU) called DelayedADURequest (`DADUReq`) to the group. The transport protocol instances of A and B do not deliver this packet immediately but buffer it instead. At the same time, a (givethrough-)timer is set. This timer expires after a time which depends on both a random component and on the network delay between the sender of the `DADUReq` and each recipient. For calculation details see [2]. In Figure 1, the timer of B expires first, triggering the delivery of the `DADUReq` to the digital lecture board of B. The information received is then used to decide if B is able to act as server for the requested data. This is the case if B has a copy of the requested data, and if B is not a late join server for another late join client. In the given example, both conditions hold true, allowing B to become the late join server for C.

B answers the request by sending a DelayedADUResponse (`DADURsp`) that includes its own IP address and port number to the multicast group. The receipt of the `DADURsp` deletes the corresponding givethrough-timer of

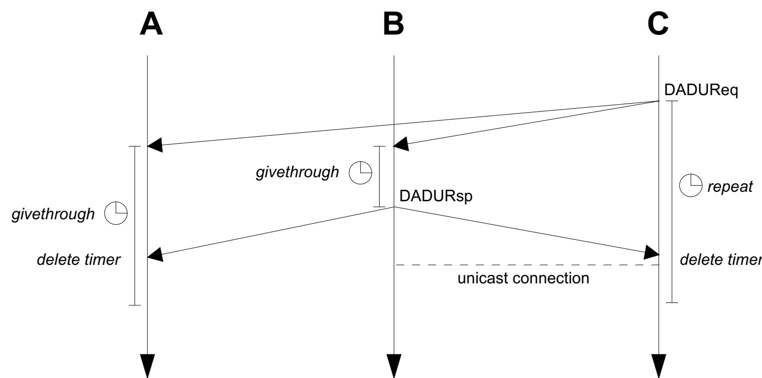

Fig. 1. Anycast mechanism

participant A, preventing the delivery of the DelayedADURequest to A's dlb. C receives the `DADURsp` as well, indicating the successful selection of a server.

In order to guarantee that this process of selection will also work if packets are lost, the sender of the `DADUReq` uses a (repeat-)timer. If this timer expires, the `DADUReq` will be retransmitted and the repeat-timer will be reset until a late join server has been identified or final failure is accepted.

Since a server is selected dynamically, a client can have different servers for different requests and a server can respond to requests from different clients. A participant can also act as a server while still being a client.

The algorithm described above makes it likely that only one response per request will be sent, regardless of the group size, and that the selected server will be a participant with a relatively small delay to the client. Since it cannot be guaranteed that the client will receive exactly one DelayedADUResponse, only the first one is processed by the client. Based on the exchanged transport addresses, a unicast connection between client C and server B is established for the transport of the late join data.

Step 2: Transmission of Late Join Data. The client is now able to request all data that are necessary to follow an ongoing session. The first information requested by the client is a survey of all pages (page list) created since the beginning of the session. Next, the client requests the data (i.e. all graphical objects) for the currently displayed page. The server answers with the transmission of all objects for that page, enabling the client to display the active page. The client is now in possession of all information that is essential to participate in the session. At this time, the dlb of the latecomer does not need the content of other pages because they are not visible. Therefore, the unicast connection can be closed. The activation of a page that has not been requested yet starts another late join cycle.

The session is not interrupted during the late join process. Therefore objects of the active page might be modified while a late join is in progress. In order to attain a consistent state, all operations which happen during a late join are buffered by the late join client. Once the transfer of late join information is complete, these modifications are applied to the active page.

In rare cases a late join server might have missed a previously transmitted packet that was not repaired before the late join data was transmitted. In this case the late join server will transmit an inconsistent state. While the late join server will eventually receive the missing information from the reliable multicast transport protocol, the late join client will need additional help to discover that it should repair the inconsistent state of the active page. The required help is provided by the late join server which appends the highest object sequence numbers of all active senders for the current page in the page information. In the event of inconsistency the late join cycle is repeated.

Analysis. The use of an additional connection ensures that no participant other than the client and the server is involved in handling the actual data transmission of the second step. Network load and processing time for the first step can be neglected, being the equivalent effort of the SRM repair mechanism for a single lost ADU. Therefore, network load and processing time applied to the multicast group as a whole are optimized. Consistency can be easily achieved through the mechanism described above. Requesting only the currently visible page effectively minimizes the initialization delay for the client. Since older pages are requested only on demand, network and application load will be spread over a relatively long time compared to a request for all pages at once.

At the same time, this approach will result in the loss of data if the last participant who possesses information about a certain page leaves the session. Experiments have also shown that the use of unicast connections can lead to major problems: first, since a participant acting as a server can establish only one unicast connection at a certain

time, it is blocked during an ongoing late join. If there are more clients than servers, some clients will have to wait for a free connection, which will increase initialization delay. Second, the anycast mechanism described above cannot guarantee that only one server will respond to a DelayedADURequest. Since a responding server opens a unicast connection in anticipation of the client's requests and the client reacts only to the first DelayedADUResponse, one or more additional servers will be blocked until a time-out has closed the connection. This can cause a further delay for other clients. Third, unicast prevents more than one client from making use of a particular transmission, although the selective requesting of pages on demand is likely to lead to more than one client needing a particular page at the same time. Fourth, opening and closing the unicast connection is costly and generates an overhead that is not to be underestimated, especially if the effectively transmitted amount of data is small. To summarize, our first approach had very promising elements, while it proved to be fragile and error prone due to the use of unicast connections to transmit the late join data.

4.2 Selective Late Join based on Multicast Transmission

In order to overcome the weaknesses of the first approach, the second algorithm uses a dedicated late join multicast group for the transmission of late join information. Participants joining an ongoing session stay members of this group until they have received the state of all pages. The address of the late join group can either be algorithmically derived from the address of the base group, or it can be announced together with the base address, e.g. by using the Session Announcement Protocol SAP [8]. The key benefits of using a dedicated multicast group are as follows: first, participants that join an ongoing session simultaneously (i.e. within a short period of time) can profit from ongoing transmissions of state information, i.e. the network load is further reduced. Second, the late join server can be stateless, which avoids blocking. Third, the time-consuming establishment of unicast connections is no longer required.

The selective late join based on multicast transmission also relies on the anycast service of the underlying transport protocol in order to select only one late join server within the multicast group. But in contrast to the first approach, we are not transmitting the IP and the port number as the payload of the `DADUReq` packets but only the type of required information (general initialization information, list of pages, objects of a certain page). The late join server selected by the anycast mechanism replies by sending the requested data directly to the dedicated multicast late join group. This scheme allows us to avoid the two distinct phases of the first approach.

Functionality of the Late Join Client. The behavior of the late join client is described by the finite state machine depicted in Figure 2. We are use the following notation:

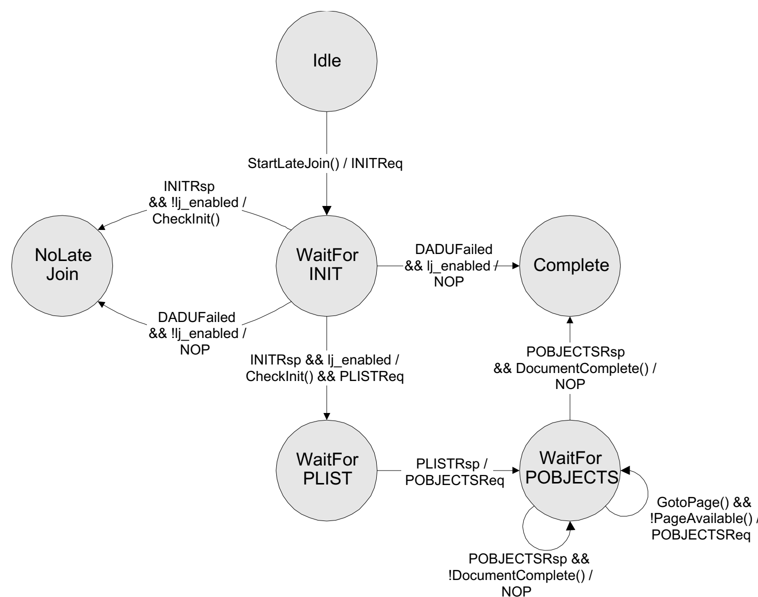

Fig. 2. Finite state machine of the late join client

`StartLateJoin()`	function that initiates the late join algorithm
`INITReq`	outgoing `DADUReq` packet to request initialization information from an arbitrary late join server
`INITRsp`	incoming response of the late join server carrying initialization information
`lj_enabled`	boolean value that determines whether or not a late join is desired by the user
`CheckInit()`	function that processes initialization values
`DADUFailed`	anycast service indicates that the anycast mechanism failed
`NOP`	no operation
`PLISTReq`	outgoing `DADUReq` to request a page list from an arbitrary late join server
`PLISTRsp`	incoming response of the late join server carrying the current page list
`POBJECTSReq`	outgoing `DADUReq` to request the objects of a certain page from an arbitrary late join server
`POBJECTSRsp`	incoming response of a late join server carrying the objects of a certain page
`DocumentComplete()`	boolean function that checks whether or not the complete online document is available
`GotoPage()`	function called by the user to change the current page of the online document
`PageAvailable()`	boolean function that checks whether or not objects of a certain page are available.

Note that outgoing DADUReq packets are transmitted by means of the regular multicast group using the anycast mechanism while incoming packets are received from the dedicated multicast late join group, i.e. the late join server always sends its replies to the late join group.

The late join client can be described by the following states: *Idle*, *NoLateJoin*, *WaitForINIT*, *WaitForPLIST*, *WaitForPOBJECTS*, and *Complete*. At the beginning the whiteboard application of a latecomer is in the *Idle* state. Calling the function `StartLateJoin()` initiates the late join mechanism; a `DADUReq` carrying an `INITReq` is sent to the multicast group. This packet can be used to request application dependent initialization information from an arbitrary late join server. The digital lecture board, for instance, uses this mechanism to request a list of all participant identifiers prior to choosing its own unique identifier.

If the anycast mechanism for the `INITReq` packet fails, i.e. an appropriate late join server is not available, the application is informed by a `DADUFailed` indication of the underlying transport service. If the late join is disabled, the client changes to the *NoLateJoin* state. Otherwise the client assumes itself to be the only member of the multicast group and changes to the *Complete* state.

Upon receipt of an `INITRsp` packet the client processes the initialization data by calling `CheckInit()`. If the late join is disabled, the client changes to the *NoLateJoin* state. Otherwise, the client assumes that a late join is required and changes to the *WaitForPLIST* state by requesting a list of all the pages of an online document through the `PLISTReq` packet delivered by the anycast mechanism. After having received the page list (`PLISTRsp`), the client requests the objects for the active page by emitting a `POBJECTSReq` through the anycast mechanism and changes to the *WaitForPOBJECTS* state. In this state, incoming `POBJECTSRsp` packets usually do not change the client's state anymore except if the complete online document has been received. In this case, the client performs a state transition to *Complete*. If a remote or local user changes the active page of an online document in the *WaitForPOBJECTS* state and the new page is not yet available, the late join client requests the objects for this page by sending another `POBJECTSReq` packet by means of the anycast mechanism.

Functionality of the Late Join Server. The late join server is stateless. If the requested data is available, incoming information requests are answered immediately by sending the data directly to the dedicated late join multicast group. Otherwise, information requests are ignored. The capability of answering incoming requests depends on the client state of the late join server. It is important to emphasize the dynamic role of the late join server: each late join client can become a temporary late join server if the client holds the requested state information. Table 1 describes the functionality of the late join server depending on the client's state where 0 indicates that a request can not be answered and 1 means that the requested data is available.

Analysis. The selective late join based on multicast retransmission eliminates the main disadvantages of the unicast solution. Participants who join an ongoing session simultaneously (i.e. within a short period of time) can profit from ongoing transmissions. The time-consuming establishment of a connection in the first approach is no longer required. Hence, *the initialization delay* is reduced significantly. The stateless server makes the scheme very *robust* and immunizes against blocking. *Consistency* is achieved through the same mechanism as in the unicast case.

Compared to the first approach, the only drawback to using a dedicated multicast group for the transmission of late join data is a slightly increased *application load.* Since clients will stay members of the late join group as long as the late join is not complete, they might receive transmissions even though they already have the information.

Table 1. Late join server functionality

client state	*INITReq*	*PLISTReq*	*POBJECTSReq*
Idle	0	0	0
NoLateJoin	0	0	0
WaitForINIT	0	0	0
WaitForPLIST	1	0	0
WaitForPOBJECTS	1	1	0/1[a]
Complete	1	1	1

a. The request can be satisfied if the objects of the page are available.

A straightforward solution would be to leave the late join multicast group as soon as the late join of a single page is complete. However, in this case the participant is obliged to rejoin the group whenever a page, that is not available at the client, is requested. The required connection establishment increases latency again. So designing late join algorithms is always a trade-off between initialization delay, network load and application load while considering application specific requirements. Experiments have shown that for a whiteboard the additional application load is not significant, while the reduction in network load can be rather high (especially when an already existing page is activated).

Another interesting question is whether or not the late join server should be a member of the late join multicast group. In our current solution the late join server sends to the late join group without being a group member. Depending on the multicast routing algorithm, this might induce a certain overhead. In cases where this overhead becomes significant, the late join server could join the late join session and respond to future requests. After a certain time-out period without further requests, the server would leave the late join group.

The anycast mechanism is used effectively to suppress multiple responses to a single late join request. However, multiple late join requests at the same time will still result in the same number of answers (number of selected servers) even though only a single server would be required. Therefore, the same suppression mechanism as for the reply implosion can be used to avoid a late join request implosion.

4.3 Generalization

The described late join algorithm based on multicast transmission can be easily modified for arbitrary replicated applications other than whiteboards given that those applications use hierarchical structures for the replicated data. The functionality of the late join server remains unchanged: requests are answered if the requested data is available, otherwise they are ignored. The late join client increases in complexity the higher the level of hierarchy. Consider n the number of hierarchy levels and D_i, $i = 0, ..., n$, data of level i. The generalized state automaton of the late join client is depicted in a simplified version in Figure 3.

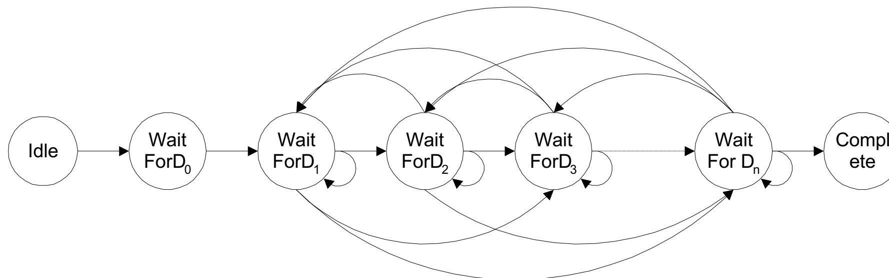

Fig. 3. Generalized late join client state machine

After having requested data for level 0, the client has to deal with data requests on arbitrary levels by both the local user and the remote participant. Therefore, we need a complete mesh for the state automaton at level 1. As soon as the data for one level is completely received, there are no more state transitions to the higher level. The states $WaitForD_0$ and $WaitForD_1$ equal the states *WaitForPLIST* and *WaitForPOBJECTS* of the whiteboard late join client depicted in Figure 2. Take a distributed text editor as an example of an application with a higher number

of hierarchy levels named, for instance, *WaitForDocument*, *WaitForChapters*, *WaitForSections*, *WaitForSubsections*, and *WaitForParagraphs*.

5 Conclusion

An efficient state initialization by means of a late join algorithm is a crucial point in the design of interactive applications based on replicated architectures. In this paper we have presented two late join mechanisms for interactive whiteboards. Our approach is located at the application level and therefore allows for a selective retransmission of the current application state. This reduces initialization delay on the one hand and makes the approach independent of the underlying transport service on the other hand. The first late join algorithm described relies on unicast connections for the initialization of a latecomer. The key issue is to avoid an implosion of replies to a single late join request by applying an anycast mechanism. This reduces network and application load significantly. The second approach uses the same avoidance mechanism but additionally improves the first scheme by using a dedicated multicast late join group for retransmission. The key benefits are: reduced network load since participants joining a session simultaneously can take advantage of ongoing retransmissions, a stateless server that makes the scheme very robust, and the avoidance of time-consuming connection management. We have discussed how the scheme can be further improved and have given indications on the generalization for arbitrary replicated applications using hierarchical data structures. Our approach has been successfully integrated into the digital lecture board. This whiteboard is currently used for synchronous teleteaching scenarios at a number of universities in Germany [4].

One of the most important issues in future research will be a quantitative analysis of the efficiency of the proposed scheme. Simulations will be used to tune the trade-off between initialization delay, network load and application load. Moreover, it would be rather interesting to analyze the influence of transmission data granularity on the efficiency of the scheme, e.g. in the case of whiteboards a late join algorithm could also request single graphical objects instead of complete pages in order to reduce network and application load.

References

1. Clark, D., Tennenhouse, D.: „Architectural Considerations for a New Generation of Protocols“. In: *Proc. ACM SIGCOMM’90*, pp. 201–208, September 1990.
2. Floyd, S., Jacobson, V., McCanne, S., Liu, C.-G., Zhang, L.: “A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing”. In: *Proceedings of ACM SIGCOMM '95*, Boston, MA, pp. 342-356, September 1995.
3. Geyer, W.: “Das digital lecture board - Konzeption, Design und Entwicklung eines Whiteboards für synchrones Teleteaching”. Dissertation (in German), Lehrstuhl Praktische Informatik IV, University of Mannheim, 1999.
4. Geyer, W.: “The digital lecture board”. Homepage of the digital lecture board project, URL: *http://www.informatik.uni-mannheim.de/~geyer/dlb/dlb.eng.html*, 1999.
5. Geyer, W., Effelsberg, W.: “The Digital Lecture Board - A Teaching and Learning Tool for Remote Instruction in Higher Education”. In: *Proc. of ED-MEDIA’98*, Freiburg, June 1998.
6. Graham, T.C.N., Urnes, T., Nejabi, R.: “Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware”. In: *Proceedings of UIST '96*, Seattle, 1996.
7. Gruman, M.: “Entwurf und Implementierung eines zuverlässigen Multicast-Protokolls zur Unterstützung sicherer Gruppenkommunikation in einer Teleteaching-Umgebung”. Master's Thesis (in German), Lehrstuhl Praktische Informatik IV, University of Mannheim, 1997.
8. Handley, M., Perkins C., Whelan E.: “Session Announcement Protocol”, Internet Draft, Internet Engineering Task Force, draft-ietf-mmusic-sap-v2-01.txt, 1999, work in progress.
9. Tung, T.-L.: “MediaBoard: A Shared Whiteboard Application for the MBone”. Master's Thesis, Computer Science Division (EECS), University of California, Berkeley, CA 94720-1776, January 1998.