A Generic Late-Join Service for Distributed Interactive Media

Jürgen Vogel, Martin Mauve, Werner Geyer, Volker Hilt, Christoph Kuhmünch,
Praktische Informatik IV
University of Mannheim
L15, 16
68131 Mannheim
{vogel,mauve,geyer,hilt,cjk}@informatik.uni-mannheim.de

ABSTRACT

In this paper we present a generic late-join service for distributed interactive media, i.e., networked media which involve user interactions. Examples for distributed interactive media are shared whiteboards, networked computer games and distributed virtual environments. The generic late-join service allows a latecomer to join an ongoing session. This requires that the shared state of the medium is transmitted from the old participants of the session to the latecomer in an efficient and scalable way. In order to be generic and useful for a broad range of distributed interactive media, we have implemented the late-join service based on the Real Time Application Level Protocol for Distributed Interactive Media (RTP/I). All applications which employ this protocol can also use the generic late-join service. Furthermore the late-join service can be adapted to the specific needs of a given application by specifying policies for the late-join process. Applications which do use a different application level protocol than RTP/I may still use the concepts presented in this work. However, they will not be able to profit from our RTP/I base implementation.

Keywords

Distributed Interactive Media, Late-Join, Generic Service, RTP/I

1. INTRODUCTION

The term *distributed interactive medium* is used to denote a networked medium which involves user interactions with the medium itself [9]. Typical examples of distributed interactive media are shared whiteboards [3][14], networked computer games [2], and distributed virtual environments [5]. One fundamental problem that applications for these media need to address is the support for participants who arrive late and wish to join an ongoing session. This is known as the *late-join problem*.

The late-join problem is challenging since distributed interactive media typically employ a *replicated distribution architecture*. That is, the application of each participant maintains a local copy of the medium's shared state. For example, in a shared whiteboard application this state may comprise the text shown on each whiteboard page as well as annotations and modifications that have been made by participants over the course of the session. Without information about the current shared state of the medium it is not possible to participate in a session. The application of a latecomer therefore needs to take special actions to retrieve the relevant parts of the shared state information.

In general a solution to the late-join problem has to perform two tasks:

1. It must identify those pieces of the shared state that are needed by the latecomer to participate in the ongoing session.
2. It needs to provide the late coming application with this information at the appropriate point in time in an appropriate way.

The first task is important since a large part of the medium's shared state may not be immediately relevant for a latecomer. In a shared whiteboard session, for example, only the state of the current page may be required to enable a latecomer to participate in the session. The state of other pages may be needed only when they become visible later on. A solution to the late-join problem therefore needs a way to explore which pieces of shared state are available in a session, and under which conditions a certain piece of the shared state is required.

Once it has been decided when the pieces of shared state are needed the second task is to retrieve those pieces at the appropriate time. This is a complex task since pieces of the shared state may be required at different times, e.g. immediately (the current page of a shared whiteboard), or later on triggered by some user action (an old shared whiteboard page that becomes visible). Furthermore the state information needs to be retrieved in a way that does not overload the network or the participating applications. This is especially difficult since distributed interactive media often involve large groups of users.

A solution to the late-join problem is called a *late-join algorithm* or *late-join service*. The application program of the latecomer is a *late-join client*, while those applications that transmit state information to the late-join client are

called *late-join servers*. The role of late-join clients and late-join servers is dynamic, i.e., a late-join client may become a late-join server for another late-join client later on.

In this paper we present a generic late-join service. It can be used for arbitrary distributed interactive media. We implemented the service using the information provided by the Real Time Application Level Protocol for Distributed Interactive Media (RTP/I) [10]. RTP/I captures the common aspects of distributed interactive media and thereby enables the development of reusable functionality without any medium-specific information. The concepts presented in this paper are independent of RTP/I and may also be used for applications that use other application level protocols. However, these application will not be able to profit from our implementation of the late join service.

The generic late-join service is highly customizable to the needs of diverse applications. Furthermore it minimizes the burden that is placed on the network and the participants of a session. The generic late-join service has been implemented in Java and is currently used by a 3D telecooperation application. Other applications are being converted to use RTP/I and the late-join service.

The remainder of this paper is structured as follows. In Section Two existing approaches to solve the late-join problem are examined. In Section Three we present our media model for distributed interactive media. This model allows us to discuss the late-join problem in a media-independent way. A short presentation of the Real Time Application Level Protocol for Distributed Interactive Media is given in Section Four. The generic late-join service is discussed in detail in Section Five. The sixth section contains a summary of our experiences with integrating the generic late-join service into an existing 3D telecooperation application. This paper is concluded by a summary and an outlook to future work.

2. EXISTING APPROACHES

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

Representatives of the first category are reliable multicast transport services that offer late-join functionality. An example of such a protocol is the Scalable Reliable Multicast protocol (SRM) [1]. A reliable multicast protocol can offer the late-join service by using its loss recovery mechanism to supply the late-joining application with all data packets missed since the beginning of the session. The application then reconstructs the current state from these packets.

The usage of transport protocol functionality to solve the late-join problem has four major drawbacks:

1. It is inefficient since a large part of the transmitted information may no longer be relevant. For example an image on a shared whiteboard page which has already been deleted in the meantime.
2. It is generally more efficient to transfer state information than to transmit all transport packets that have lead to that state. When editing a text, for example, it makes sense to transmit the state of the text rather than all the packets that contain the description of a character that has been typed or deleted. This becomes even more important when the overhead for packet headers is taken into account.
3. The application either has to be able to reconstruct every packet that has ever been transmitted, or the transport service needs to buffer the transmitted packets indefinitely. This is clearly not acceptable for a large number of applications.
4. The state of certain media may not be easily reconstructible from a simple replay of packets. The problem here is that for certain media (such as an networked action game) an event is only valid at a certain point in time. In order to reconstruct the state of such a medium from outdated packets, the application would have to perform a timewarp to the beginning of the session and then a rapid replay of states and events. It is by no means guaranteed that all, or even a significant number of, application will 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 exists that is able to act as the late-join server for the shared state. When a late coming application joins the session it may contact the state server which will in turn deliver the relevant state information. An example where a centralized state server is used for late-join purposes is the Notification Service Transport Protocol [12].

A centralized 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. Because of these drawbacks we have decided not to use a centralized late-join server for our generic late-join service.

Distributed late-join approaches seek to avoid the problems of a centralized approach by involving multiple applications in the late-join process. In particular, many applications may be able to assume the role of a late-join server for any given piece of shared state. The failure of any single application can generally be tolerated without preventing a latecomer from joining the session. Applications that use a distributed late-join approach are the network text editor (NTE) [6] and the digital lecture board (dlb) [3].

The innovative idea of the method used for the dlb is to employ a separate (multicast) group for the data that is

transmitted to latecomers. Requests for the state of a page are transmitted to the regular session by the late-join clients. Replies are sent to the late-join group by the applications that act as late-join servers. An application may leave the late-join group once it has received all required information about the shared state. A reply implosion of the potential late-join servers is prevented by using an anycast mechanism that is based on random timers.

The approach used for the dlb has several positive characteristics. First, it limits the burden of the late-join activity that is placed on session participants who are not latecomers. These participants just need to check whether they have been selected as late-join servers. The transmitted state information is only received by those applications that have not yet finished their late-join activity. Second, for the same reason, the approach also minimizes the network load: state information is only transmitted over those parts of the network that lead to an application still lacking late-join information. Third, the dlb late-join effectively prevents a reply implosion when more than one participant is able to act as a late-join server.

However, there are also some areas in which the dlb solution can be further improved:

1. In addition to preventing a reply implosion, a request implosion should also be avoided. This is particularly important if a user action (such as changing the page in a shared whiteboard) may trigger the need for additional state information. In this case multiple late-join clients might require the same state information simultaneously.
2. Some applications that represent potential late-join servers should join the late-join group. Requests for state information could then be transmitted to the late-join group, so that uninvolved applications do not have to handle requests for state information. Only as a fallback solution should the request be sent to the original group. This requires to give criteria that decide which potential late-join servers should enter or leave the late-join session.
3. The dlb approach is application-dependent. It is not based upon a generic application level protocol, and it is not easily customizable to the diverse needs of different distributed interactive media.

Because of its positive characteristics we have chosen the dlb approach as the basis for our generic late-join service. Upon this basis we have developed an improved and generic late-join service for distributed interactive media.

3. MEDIA MODEL

In order to provide a generic service that is reusable for a whole class of media it is important to investigate the media model of this media class. In the following we give a brief overview of the characteristics of the distributed interactive media class. A more detailed discussion can be found in [9].

States and Events

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

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

Any state change that is not a fully deterministic function of time is caused by an *event*. Generally events are (user) interactions with the medium, e.g. the user makes an annotation on a shared whiteboard page. Whenever events occur, the state of the medium is in danger of becoming inconsistent. Therefore, an event usually requires that the applications exchange information - either about the event itself or about the updated state once the event has taken place.

Partitioning the Medium - Sub-Components

In order to provide for a flexible and scalable handling of state information it is desirable to partition an interactive medium into several *sub-components*. In addition to breaking down the complete state of an interactive medium into more manageable parts, such partitioning allows the participants of a session to track only the states of those sub-components in which they are actually interested. Examples of sub-components are 3D objects (an avatar, a car, a room) in a distributed virtual environment, or the pages of a shared whiteboard. Events affect only a *target* sub-component. Sub-components other than the target are not affected by an event.

4. RTP/I

While the media model provides an important insight into the distributed interactive media class, the design and implementation of a generic late-join service requires a more formal foundation. The Real Time Application Level Protocol for Distributed Interactive Media (RTP/I) [10] provides such a foundation. While our implementation of the generic late join service makes use of RTP/I, the concepts presented here may also be used by applications which do use other application level protocols.

RTP/I consists of two parts: a data transfer protocol for the transport of event and state information, and a control protocol for meta-information about the medium and the participants of a session:

- The *data transfer protocol* (RTP/I) frames the transmitted states and events of the medium with information that is common to the distributed interactive media class. With this information a generic service, like the late-join service presented here, can interpret the semantics of the information without knowing anything about the medium-specific encoding. Typical examples for the information contained in the RTP/I data framing are a timestamp which indicates at what time an event happened or a state was calculated, an identifier for the affected sub-component and the type of the data (event vs. state information). In addition to state and event transmission RTP/I is also used to request the state of a sub-component in a standardized way.
- The *RTP/I control protocol* (RTCP/I) conveys information about the participants of a session. This includes the participants' names and email addresses. This information can be used to establish a light-weight session control. Moreover RTCP/I provides information about the sub-components that are present in a session. Information about each sub-component is regularly announced. The announcement contains three types of information: (1) the ID of the sub-component, as it is also used in the framing of the data transfer protocol. (2) An application level name for the sub-component. This name allows an application to identify the sub-component. A typical example for an application level name could be the title of a shared whiteboard page. (3) An announcement whether this sub-component is actively used by any participating application in order to present the medium to the user. The visible pages of a shared whiteboard presentation would belong to the class of active sub-components, while the invisible pages would belong to the class of passive sub-components. All three types of information can be used by the late-join service and the application using the late-join service to determine the relevance of sub-components.

RTP/I is closely related to the Real Time Transport Protocol (RTP) [7] which is mainly used for the transmission of audio and video. However, while RTP/I reuses many aspects of RTP it has been thoroughly adapted to meet the needs of distributed interactive media.

5. GENERIC LATE-JOIN SERVICE

The architecture of the generic late-join service is depicted in Figure 1. The late-join service intercepts the data (events, states, and requests for states) that arrives from the base session. Since this data is transmitted using RTP/I the generic late-join service can understand the semantics of this data to a degree sufficient to provide the late-join functionality. Knowledge about the medium-specific encoding is not required. After examining the data the late-join service forwards it to the application.

The application transmits all regular data directly to the base RTP/I session without informing the late-join service. late-join information is handed from the application to the late-join service. An example for this type of information is the state of a sub-component that is required by the late-join service to support a remote latecomer. The reason for passing this data to the late-join service instead of transmitting it over the base RTP/I session is as follows: the generic late-join service maintains an additional late-join RTP/I session. This session is used to transmit all late-join oriented data. The late-join service joins and leaves this additional RTP/I session at appropriate times. This ensures that only a small subset of all participants need to handle late-join data.

Finally there exists a generic services channel. This channel is shared by all generic services (there may be multiple generic services present in a single application). It is used to convey signalling data for the generic services. In order to limit routing effort at the network level the generic services channel has the same network address as the base RTP/I session. Transport layer multiplexing is used to separate the generic services channel from the base RTP/I session. An application remains a member of the generic services channel for the lifetime of a session.

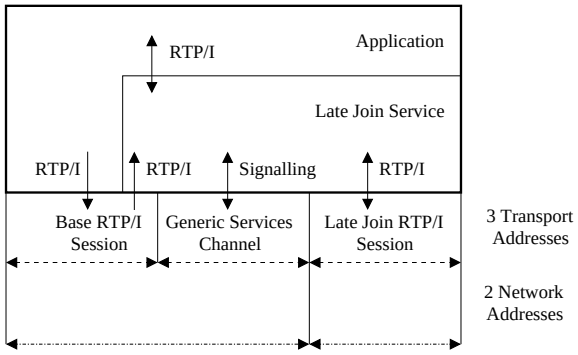

Figure 1: Architecture of the late-join service

When joining an ongoing session the late-join service will learn about the sub-components that are present in a session through the RTCP/I reports on sub-components. Whenever a new sub-component is detected the late-join service informs the application and requests information on how the late-join should be performed. The application may choose between a set of policies, ranging from no action to immediate retrieval of the sub-component's state. In addition to the sub-component ID the application may use the information which is delivered via RTCP/I (application level name, and whether the sub-component is active for at least one user) to determine which policy is appropriate for a given sub-component.

When the condition occurs that was specified by the policy the state of the sub-component is requested by the late-join service. This is done by transmitting RTP/I state queries to the late-join group using a message implosion avoidance mechanism. This mechanism makes sure that no message implosion occurs if multiple latecomers want to request the state of the same sub-component at the same time. A similar mechanism was first used by SRM [1] to achieve a scalable reliable multicast. In order to avoid a message implosion the late-join service waits a random time before transmitting an RTP/I state query. The value of the random timer is evenly distributed in an interval that depends on the distance (in terms of network delay) to the participant who transmitted the report of the sub-component. The smaller the distance, the smaller is the upper bound on the timer. Therefore it is likely that applications which are close to the

origin of a report will reply first. Any other late-join service that wants to send a state query for the same sub-component suppresses this message when it sees that the message has already been transmitted by someone else. In this way a message implosion can be prevented effectively.

The request is repeated if there is no answer after a certain amount of time. If multiple requests for the sub-component's state remain unanswered, it is concluded that there is no late-join server for that sub-component present in the late-join group. In this case the late-join service uses the generic services channel to request that a participant which holds the state of the sub-component joins the late-join session and transmits the state. If this fails, too, then the application is informed.

When the late-join service receives the desired state information it passes it on to the application and marks this sub-component as complete. When there are no new sub-components detected for a period of time and all sub-components are marked as complete, then the late-join is finished. At that time the late-join service may leave the late-join group. If, at any later point in time, a new sub-component is detected, then the application may ask the late-join service to resume its duty and join the late-join group again. When new sub-components are introduced in an ongoing session this can be used to conveniently request the state of the new sub-components in a way that is policy-driven.

Late-Join Policies

An application that uses the generic late-join service may specify a late-join policy for each sub-component that has been discovered by the late-join service. Setting different policies for sub-components makes it possible to retrieve different sub-components with different priorities; some may be retrieved immediately, others when network capacity is available or when they become important for the presentation to the user. The use of policies ensures high flexibility and an easy adaptation to the needs of individual applications. Existing solutions to the late-join problem, such as the one used for the digital lecture board, lack this ability.

The generic late-join service offers five late-join policies:

- event-triggered late-join,
- no late-join,
- immediate late-join,
- network-capacity-oriented late-join, and
- application-initiated late-join.

An application may change the late-join policy for a given sub-component at any time. We will examine the event-triggered late-join policy in more detail while the other policies are only outlined briefly.

Late-Join Policy: Event Triggered Late-Join

An application may decide that a sub-component is required only when it is the target of an event. For example, if a page in a shared whiteboard becomes the active page by means of an "activate page" event. This is supported through the event-triggered late-join policy. Besides deferring the request of state information until it is actually needed, this policy also increases the likelihood that multiple latecomers may profit from a single state transmission if multicast is used. The reason for this is that the late-join service may refrain for a long time (until the first event for the sub-component occurs) from requesting the state of a sub-component with an event-triggered late-join policy. All latecomers who join during that period will profit from a single transmission of the sub-component's state.

The finite state machine for the event-triggered late-join policy is shown in Figure 2. The name of the initial late-join state is "sub-component discovered" (SD). At the time the application chooses the late-join policy the late-join state changes to "wait-for-event" (WFE). In the WFE state the late-join service listens to incoming RTP/I packets and checks whether they contain an event or a state for the sub-component. If a state is received then the state is handed to the application, and the late-join state becomes "complete" (COMP) for the sub-component. This may happen when a state is transmitted for some reason (e.g., as a means of resynchronization) in the base RTP/I session.

When an event for the sub-component is received, then the late-join state for the sub-component becomes "sub-component needed" (SN), and a random timer is set. The random timer prevents a request implosion in the case that the sub-component is required by multiple latecomers at the same time. If the state of the sub-component is received while the request timer is running a transition from SN to complete (COMP) is performed.

When the random timer expires before the state of the sub-component has been received, then an RTP/I *state query* packet is transmitted to the late-join group, another random timer is set, and the late-join state changes to "sub-component requested I" (SR I). When a reply to this request is received then the late-join is complete for that sub-component. If no reply is received before the timer expires, then a new request is transmitted. In the case that a remote request for the sub-component is received, the timer is reset (not shown in Figure 2).

If multiple requests fail, then there is not appropriate late-join server for this sub-component in the late-join group. In this case the late-join state for this sub-component changes to "sub-component requested II" (SR II), and a join request is transmitted on the generic services channel. Upon receiving this request potential servers check whether they should join, using a join implosion avoidance mechanism similar to the one used for request implosion avoidance. When an application decides to join, it also transmits the required information to the late-join group.

When repeated requests for state information fail, the application is informed, and the state is set to WFE. A new event for the sub-component triggers another request round. This makes sense since an event indicates that the problem has been repaired and that a late-join server for this sub-component should be available.

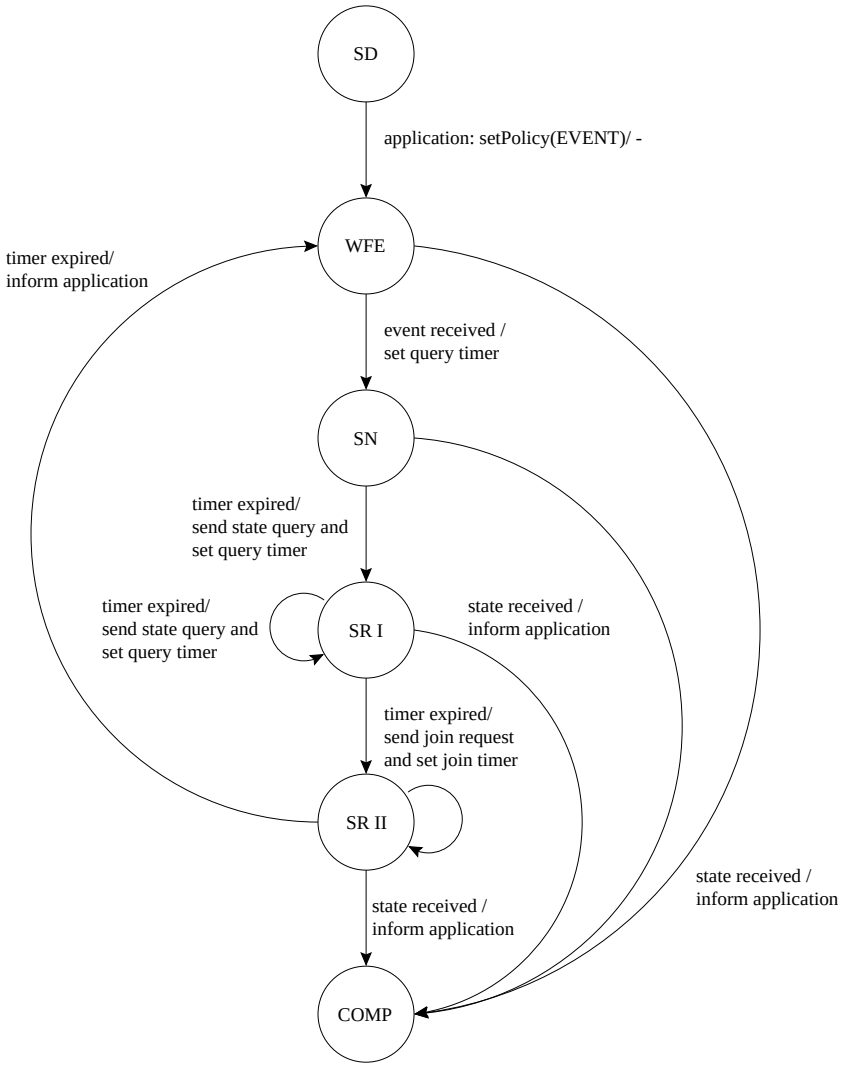

Figure 2: Event-triggered late-join

Late-Join Policy: No Late-Join

This policy is chosen by the application to indicate that it is not interested in the sub-component. In a distributed virtual environment this policy could be used for sub-components 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. This has a positive effect on the initialization delay as well as on the network and the application load. When the late-join service is notified that the application has chosen this policy for a sub-component, then the sub-component is marked as complete.

Late-Join Policy: Immediate Late-Join

An application may choose the immediate late-join policy when the sub-component is required at once to present the medium to the user (e.g., the currently visible pages of a shared whiteboard). An application can derive the information whether a sub-component is required immediately from the sub-component ID, from the application level name, or from the fact that it is needed to display the medium to at least one user.

The finite state machine for the immediate late-join policy is similar to the one of the event triggered late-join policy. The main difference is that there exists no wait for event (WFE) state. Instead the late-join state becomes "sub-component needed" as soon as the application sets the immediate late-join policy for a sub-component.

Late-Join Policy: Network-Capacity-Oriented Late-Join

For sub-components where the state is not immediately required an application 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 application. This replaces the WFC state from the event-based late-join policy. If the traffic falls below a threshold set by the application then a transition to the "sub-component needed" state is performed. Whenever a state query is about to be transmitted, the network traffic is checked. Only when it is still below the threshold, is the

query actually transmitted. Otherwise the query is delayed further. In all other aspects the network-capacity-oriented late-join policy is identical to the event-triggered late-join policy.

Late-Join Policy: Application-Initiated Late-Join

At any point in time the application may choose to change the late-join policy of a sub-component. In this way it is possible to upgrade policies like "no late-join" or "network capacity oriented late-join" to any other policy if this becomes necessary.

The application can define new late-join policies by initially setting the late-join policy for a sub-component to "no late-join". When the application-defined policy indicates that the state of the sub-component should be retrieved then the application can change the policy to "immediate late-join". This should be used only for experimental purposes. If another late-join policy becomes important for certain applications, then the late-join service should be expanded to include it.

Joining and Leaving the Late-Join Session

Unlike existing approaches, the generic late-join service allows a small number of applications which have completed their late-join process to stay in the dedicated late-join group. If these applications are chosen well they can assume the role of late-join servers for future latecomers, while the vast majority of applications (those that have completed their late-join process and that are not member of the late-join group) are completely uninvolved in the late-join process.

This approach raises the question of who should be member of the late-join group? Obviously all applications that have not yet finished the late-join process for all sub-components should stay in the late-join group. A late-join service could theoretically leave the late-join group as soon as the late-join for all sub-components has been completed. However, it should not do so without further consideration since this could leave the late-join group without a late-join server for certain sub-components. This would increase the time that a late-join client has to wait before it gets the state of a sub-component. It is therefore important to define an algorithm that decides which applications should stay members of the late-join group, even if they have completed their own late-join operations for all sub-components.

There are a number of criteria that need to be considered for an algorithm that decides whether an application should stay in the late-join group or not:

- **Group size**. The late-join group should be as small as possible. The smaller the group, the less network traffic is generated, and the fewer the applications that are involved in late-join management. A small group also decreases the likelihood that more than one late-join server will reply to the state query of a late-join client.
- **Sub-component presence**. Ideally each sub-component for which the state is likely to be requested should have a late-join server in the late-join group. This reduces the delay for late-join clients.
- **Group invariance**. The number of join and leave operations should be small for the late-join group since each of these operations is associated with overhead at the network layer (e.g., multicast routing).
- **Simplicity**. The applications should be able to perform the algorithm with a minimal effort in computation and communication.

Let us consider three different approaches to decide which applications should remain in the late-join group to act as late-join servers: distributed, isolated, and application controlled.

Distributed Membership Management

In distributed membership management the applications exchange information about their capabilities to act as late-join servers. This can be done via the generic services channel. With this information an optimal set of potential late-join servers can be determined. For example, the participants who are able to provide late-join server functionality for many sub-components should be preferred as members of the late-join group.

The main drawback of the distributed membership management is its complexity. Applications need to exchange additional information to allow for this kind of membership management. This information needs to be transmitted and processed, which may lead to significant overhead, especially for large sessions. For these reasons we have chosen not to use distributed membership management for our generic late-join service.

Isolated Membership Management

Isolated membership management seeks to avoid additional messages and processing overhead by using local information. Each application decides on its own whether it should join or leave the late-join group. Isolated membership management therefore seeks to increase simplicity at the cost of a slight reduction in the other quality criteria.

Our generic late-join service uses isolated membership management. Applications will leave the late-join group by means of a 'smart timeout', and they enter the late-join group upon the request of a late-join client.

An application leaves the late-join group if it has not answered any state queries for a certain amount of time. This amount of time is not fixed, but is calculated based on three values:

1. An average late-join group membership time provided by the application. In this way the application can give a hint to the late-join service on how fast applications should leave the late-join session.
2. The number of sub-components that the application can provide as a late-join server compared to the total number of sub-components present in the session. In this way applications that can serve a large percentage of the sub-components will stay longer in the late-join group.
3. The number of late-join state queries that could have been answered by the late-join server compared to the number of late-join state queries that actually have been answered by this application and not by some other late-join server. The lower this percentage is, the less impor-

tant is the presence of the application in the late-join group.

When the timer expires the application leaves the late-join group. It may happen that the late-join group contains no late-join server for a given sub-component. If there is no late-join activity for a prolonged time the late-join group may even become empty. Generally this is a good thing, since it saves resources in the event that late-joins are infrequent. However, there must also be a way to allow applications to re-join the late-join group if a new late-join client appears.

As described above, a late-join client transmits a message on the generic services channel if the state queries for a sub-component remain unanswered in the late-join group. All applications that are able to become a late-join server for this sub-component use an SRM style implosion-avoidance mechanism to decide who will actually join the late-join group. The generic service of the selected application transmits an acknowledgment to the generic services channel, joins the late-join group, and transmits the requested state of the sub-component.

Application-Controlled Membership Management

In some cases the application may want to decide explicitly who should join the late-join group rather than leaving this decision to the late-join service. For example, in a medium that uses a floor control mechanism only the floor holder may be able to transmit the state of a sub-component. Since there is only one candidate for joining the late-join session, it would be wasteful to use an implosion avoidance mechanism. Therefore our late-join service allows the application to specify that it should immediately enter the late-join group if the state of a certain sub-component is requested.

In order to determine when an application should leave the late-join session, the smart timeout mechanism described above is also used for application-controlled membership management. This is reasonable since an application will generally not be able to determine with a higher accuracy than the late-join service when it is no longer needed as a late-join server.

Generic Late-Join Service API

The interface to the late-join service is depicted in Figure 3. The first two functions are called when RTP/I and RTCP/I data is received for the original RTP/I session. Based on this information the late-join service discovers new sub-components and triggers requests for the state of sub-components.

When a new sub-component is discovered, then the late-join service asks the application about the late-join policy that should be associated with the sub-component. If all sub-components should be treated with the same policy then it is possible to set a default policy by means of `setDefaultPolicy`. The late-join service will then refrain from asking the application about late-join policies for individual sub-components.

An application can at any time call `setPolicy` to assign a new late-join policy to a sub-component. This may also be called on a sub-component for which a late-join has already been completed. The late-join may be used in this way to recover the state of sub-components in a late-join policy driven fashion.

The application may specify the policy for joining the late-join group and the base time for leaving it. When the late-join service needs to transmit the state of a sub-component as a late-join server, then it requests the sub-component's state from the application by means of the `getSubComponentState` method. Finally the application may be informed of an unsuccessful late-join attempt through a `ljFailed` call.

```
Implemented by the generic late-join service:
void rtpiDataReceived(RTPIData rtpiData)
void rtcpiPacketReceived(RTCPIPacket
                         rtcpiPacket)
void setPolicy(LJPolicy policy, long subID)
void setDefaultPolicy(LJPolicy policy)
void setJoinGroupPolicy(JGPolicy policy)
void setLeaveGroupBaseTime(long baseTime)

Implemented by the application:
LJPolicy askForPolicy(long subID)
RTPIData getSubComponentState(long subID)
void ljFailed(long subID)
```

Figure 3: Generic late-join service API

Consistency and the Generic Late-Join Service

A distributed interactive medium generally needs to take specific actions to maintain a consistent shared state for all participants of a session. This includes ensuring that events are applied to the state of the medium in the correct order at the appropriate point in time. It may also be necessary to realize state repair functionality to recover from network partitioning or lost events. The piece of software that is responsible for these actions is called a *consistency service*.

The generic late-join service provides the application and the consistency service with an initial state of the sub-components in a policy driven and efficient way. It is not the task of the late-join service to realize the functionality that should be provided by a separate consistency service. This would limit the applicability of the generic late-join service, since different distributed interactive media may have different requirements regarding consistency while having the same requirements for a late-join service. In addition to the generic late-join service presented here we have designed and implemented a consistency service for distributed interactive media which are continuous (i.e. these media require that events are applied to the state of the medium at a given point in time). A detailed description of the algorithm for this consistency service can be found in [11].

6. EXPERIENCES

We have used the generic late-join service for a 3D teleco-operation application called TeCo3D - a shared workspace for dynamic and interactive 3D models [8]. TeCo3D was developed to allow users to share collaboration-unaware VRML (Virtual Reality Modeling Language) models, i.e. models which have not been specifically developed to be

used by more than one user at a time. With this functionality it is possible to include arbitrary VRML content, as generated by standard CAD or animation software, into teleconferencing sessions.

TeCo3D was developed by reusing a Java3D VRML loader as 3D presentation and execution engine and employs a completely replicated distribution architecture with reliable multicast as means of communication. When a user imports a local VRML object, the VRML code is parsed and the parts which are responsible for user interactions are replaced with custom components turning the collaboration-unaware object into a collaboration-aware model. User initiated operations are captured by the custom components and are transmitted to all peer instance in the session, where they are injected into the local model. In order to provide access to the shared state of items on the shared workspace we have enhanced the VRML loader by a method to get and set the state of arbitrary VRML objects.

The media model for distributed interactive media provides a good fit for TeCo3D. The sub-components are the VRML objects. The state of these objects represent the state of the sub-components while the events are user interactions with the objects. TeCo3D uses RTP/I as application level protocol.

TeCo3D uses a floor control based consistency service. Like the late-join service presented in this work, the consistency service is RTP/I based and generic. The floor of a sub-component identifies the participant with a valid state of a VRML object. Only this participant is able to transmit states and events for the VRML object.

The generic late-join service has been developed completely separate from TeCo3D using a simple demo application. The integration of the generic late-join service into TeCo3D was straight forward. It took less than 5 hours to complete the integration. The following choices were made for adapting the generic late-join service to the needs of TeCo3D:

- The late-join policy for those sub-components that are currently visible for at least one participant is set to immediate late-join.
- The late-join policy for those sub-component that are not visible for any user is set to event triggered.
- The late-join group join policy is set to application defined. This is reasonable since the consistency service of TeCo3D allows only a single participant to reply to state queries for a given sub-component.

The simple and straight forward integration, as well as the easy way of adaptation shows that the generic late-join service is indeed generic and useful for distributed interactive media. In order to allow others to experiment with this service, the generic late-join service can be downloaded as Java sourcecode from our web site [13].

7. CONCLUSION AND OUTLOOK

In this paper we have presented a generic, RTP/I based, late-join service for distributed interactive media. This service enables latecomers to join and participate in an ongoing session. Since the late-join service does not use any media specific information it can be employed by arbitrary applications that use RTP/I as application level protocol.

The two main innovations of the generic late-join service are its efficiency and its flexibility. The efficiency of the generic late-join service is realized by the usage of a dedicated late-join session. Unlike than in existing approaches, only members of this session will be regularly involved in the late-join process. Members of the base session are not burdened with the late-join activity. The membership of potential late-join servers in the late-join group is controlled by means of a smart time-out and join signals from latecomers.

The flexibility of the late-join service is realized by means of diverse late-join policies for the sub-components of the distributed interactive medium. The late-join policies allow the simple tailoring to the individual needs of application. Furthermore the application can decide on the policy for joining the dedicated late-join session.

We have integrated the generic late-join service into an existing, RTP/I based, 3D telecooperation application called TeCo3D. This integration was simple and needed less than 5 hours. The source code of the generic late-join service is available for download.

Currently we are working on a C++ port of the late-join service. Furthermore we will integrate the late-join service into two additional applications: a shared whiteboard and distributed Java animations for teleteaching purposes. We expect to get important information about how to improve the late-join service from these items of future work.

REFERENCES

1. S. Floyd, V. Jacobson, C. Liu, S. McCanne and L. Zhang. A reliable multicast framework for leight-weight sessions and application level framing. In: *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, 1997, pp. 784 - 803.
2. L. Gautier, C. Diot. Design and Evaluation of MiMaze, a Multi-player Game on the Internet. In: *Proc. of IEEE International Conference on Multimedia Computing and Systems*, Austin, Texas, USA, 1998, pp. 233-236.
3. W. Geyer and W. Effelsberg. The Digital Lecture Board - A Teaching and Learning Tool for Remote Instruction in Higher Education. In: *Proc. of 10th World Conference on Educational Multimedia (ED-MEDIA) '98*, Freiburg, Germany, 1998. Available on CD-ROM.
4. W. Geyer, J. Vogel, and M Mauve. An Efficient and Flexible Late Join Algorithm for Shared Whiteboards. To appear in: *Proc. of the Fifth IEEE International Symposium on Computers and Communications, ISCC'2000*, Antibes, France, July, 2000.
5. O. Hagesand. Interactive multiuser VEs in the DIVE system. In: *IEEE Multimedia*, Vol. 3, No. 1, 1996, pp. 30 - 39.
6. M. Handley, and J. Crowcroft. Network text editor (NTE): A scalable shared text editor for the MBone. In. Proc. of the ACM SIGCOMM'97, Cannes, France, 1997, pp. 197 - 208.

7. V. Jacobson, S. Casner, R. Frederick and H. Schulzrinne. *RTP: A Transport Protocol for Real-Time Applications*, Internet Draft, Audio/Video Transport Working Group, IETF, draft-ietf-avt-rtp-new-04.txt, 1999. Work in progress.
8. M. Mauve. TeCo3D: a 3D telecooperation application based on VRML and Java. In: *Proc. of SPIE Multimedia Computing and Networking (MMCN) '99*, San Jose, CA, USA, published by SPIE, Bellingham, Washington, USA, January 1999, pp. 240 - 251.
9. M. Mauve, V. Hilt, C. Kuhmünch and W. Effelsberg. A General Framework and Communication Protocol for the Transmission of Interactive Media with Real-Time Characteristics. In: *Proc. of IEEE Multimedia Systems (ICMS) '99*, Florence, Italy, published by IEEE Computer Society, Los Alamitos, California, USA, June 1999, Vol. 2, pp. 641 - 646.
10. M. Mauve, V. Hilt, C. Kuhmünch, J. Vogel, W. Geyer and W. Effelsberg. *RTP/I: An Application Level Real-Time Protocol for Distributed Interactive Media.* Internet Draft: draft-mauve-rtpi-00.txt, 2000. Work in progress.
11. M. Mauve. *Consistency in Continuous Distributed Interactive Media*. Technical Report TR-9-99, Reihe Informatik, Department for Mathematics and Computer Science, University of Mannheim, November 1999.
12. J. F. Patterson, M. Day and J. Kucan, Notification servers for synchronous groupware. In: *Proc. of the ACM conference on Computer supported cooperative work (CSCW) ' 96*, Boston, USA, 1996, p. 122.
13. RTP/I. The RTP/I homepage. http://www.informatik.uni-mannheim.de/informatik/pi4/projects/RTPI/index.html
14. T. L. Tung. MediaBoard: *A Shared Whiteboard Application for the MBone*. Master's Thesis, University of Calafornia, Berkeley, California, USA, 1998.