An Application Developer's Perspective on Reliable Multicast for Distributed Interactive Media

Martin Mauve and Volker Hilt
Praktische Informatik IV
University of Mannheim
L15, 16
68131 Mannheim
{mauve,hilt}@informatik.uni-mannheim.de

ABSTRACT

In this paper we investigate which characteristics reliable multicast services should have in order to be appropriate for use by distributed interactive media applications such as shared whiteboards, networked computer games, or distributed virtual environments. We take a close look at the communication requirements of these applications and at existing approaches to realize reliable multicast. Based on this information we deduce which reliable multicast transport protocols are appropriate for the different aspects of distributed interactive media. Furthermore we discuss how the application programming interface of a reliable multicast service should be designed in order to support the development of applications for distributed interactive media.

Keywords. Reliable Multicast, Distributed Interactive Media.

1. INTRODUCTION

Applications for distributed interactive media such as shared whiteboards, networked computer games, and distributed virtual environments have gained importance rapidly over the recent years. The main characteristic of a distributed interactive medium is that it involves user interactions with the medium itself. Applications for distributed interactive media frequently rely on multicast communication since a session involving such a medium is typically attended by a group of more than two users. These applications often need to exchange information in a reliable way. This is necessary to make sure that the shared state of the medium remains consistent for all participants of a session. Reliable multicast is therefore one key issue for the realization of many distributed interactive media.

In this paper we examine the specific needs that applications for distributed interactive media place on reliable multicast services. It is not the aim of this work to define a specific reliable multicast algorithm. Instead we focus on the issue of which existing approaches are appropriate for distributed interactive media. Moreover, we discuss how the application programming interface for a reliable multicast service should be designed in order to support the development of applications for this media class. The findings presented in this paper are based on our experiences with developing applications for diverse distributed interactive media. These include a shared whiteboard [6], a 3D telecollaboration application [12], and distributed Java simulations for teleteaching purposes [9].

The remainder of this paper is structured as follows: In Section Two we present our model of the distributed interactive media class. This model makes it possible to understand and discuss the communication needs of media belonging to this class independent of any specific medium. In Section Three we give a brief summary of existing approaches that can be employed to achieve reliable multicast. Based on this information and on the media model the fourth section contains a discussion of which reliable multicast approaches are appropriate to be used for the different aspects of distributed interactive media. In Section Five we take a brief look at congestion control for distributed interactive media. In Section Six we investigate the design of application pro-

gramming interfaces (APIs) for reliable multicast services that wish to support distributed interactive media applications. The paper concludes with a summary and an outlook for future work.

2. MEDIA MODEL

In this section we give a definition of the term *distributed interactive media* in the form of a media model. A media model is an important tool to understand the commonalities of a media class. It allows the discussion of a whole media class instead of a single medium or application. In particular it is a good starting point to investigate the common requirements in regard to reliable multicast services.

2.1. States and Events

A *distributed interactive medium* has a *state* [13]. For example, 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 maintains a local copy of (parts of) the medium's state. Applications for distributed interactive media are therefore said to have a *replicated 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* up to the desired degree. A distributed interactive medium may tolerate brief periods of inconsistency, e.g., in order to increase responsiveness.

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 calculate the required state changes independently. 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*. Events can be separated into *external events* and *internal events*. External events are (user) interactions with the medium, e.g., the user makes an annotation on a shared whiteboard page. Internal events are non-deterministic internal changes in the state of the medium, such as the generation of a random number which determines the new heading and direction of a computer-controlled entity. 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.

Distributed interactive media can be sub-divided into *discrete* and *continuous* distributed interactive media. While discrete distributed interactive media (e.g., shared whiteboards) change their state only in response to events, continuous distributed interactive media (e.g., networked computer games) may also change their state because of the passage of time. In order to ensure consistency discrete media need to make sure that events are applied to the shared state in the correct order. Continuous media must execute events in the proper order and at the correct point in time. If these conditions are not met the state of the distributed interactive medium may become inconsistent and a repair of the shared state may be required.

2.2. Partitioning the Medium - Sub-Components

In order to provide for a flexible and scalable handling of state information, it is often 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 house, a car) in a distributed virtual environment, or the pages of a shared whiteboard. Events, external as well as internal, affect a *target* sub-component. Sub-components other than the target are not affected by an event. This guarantees the independence of sub-components and it thereby allows applications to track the state of one sub-component independent of the state of other sub-components.

2.3. Environment

While it would be conceivable to declare all the information that is required by the application to display the distributed interactive medium to be part of the medium's state, this is generally not desirable. Usually a substantial part of this information remains constant over the course of a session. We call this constant information the

environment of a distributed interactive medium. Examples of environments are the base world descriptions of distributed virtual environments, or the postscript slides of shared whiteboard presentations. Since the environment stays constant, there are no mechanisms required to synchronize it among the participants of a session - the environment information just needs to be received once by each participant.

It is therefore a good idea to make the environment part of the medium data as large as possible and to minimize the amount of state information. The distinction between state and environment information is situation dependent. A participant might choose to introduce a new postscript slide on-the-fly into an ongoing shared whiteboard presentation, thereby making this slide part of the medium's state. It is the task of the application designer to distinguish between state and environment information.

3. RELIABLE MULTICAST APPROACHES

Current state-of-the-art reliable multicast approaches can be divided into three main groups: structured receiver group, unstructured receiver group, and forward error correction (FEC) based approaches [7]. The latter group is special in that it can be used either stand-alone or in combination with any one of the other two ways to achieve reliable multicast.

3.1. Structured Receiver Group

The approaches based on a structured receiver group require that the receivers of a reliable multicast transmission are arranged in groups. In each group there exists one dedicated participant which is responsible to provide the other group members with local repair information about lost packets. The dedicated participants of all groups usually form a tree structure with the sender as the root of the tree. Acknowledgments (ACKs) about received packets and negative acknowledgments (NACKs) about missing packets are aggregated by the dedicated participants as they travel from the receiver groups towards the sender. Furthermore the dedicated participants may locally repair packet loss experienced by participants that belong to their group. Therefore the original sender of a packet is only burdened with retransmissions in a small number of cases. Examples for reliable multicast protocols that use a structured receiver group are RMTP-II [18] and TRAM [8]. An overview of these approaches is presented in [5].

The main advantage of these approaches is that, once the grouping is established, the repair of lost packets is very efficient, fast, and highly scalable. The drawback of approaches with a specific arrangement of receivers is that the receiver structure (partly) replicates the multicast-routing infrastructure at the application level. This raises the problem of accessing network-topology information from the application level and it implies a significant overhead in establishing and maintaining the structure formed by the receivers. These problems become even more significant when the group membership is dynamic, i.e., when participants join and leave the session frequently. Furthermore this approach does not cope well with large numbers of senders since the arrangement of receivers is usually sender-specific. Each sender therefore requires the management of one receiver group structure.

3.2. Unstructured Receiver Group

Approaches with an unstructured receiver group rely on the receivers to establish reliability by sending NACKs for the data that they did not receive to the whole group of participants. Since the same data may get lost for multiple receivers simultaneously a *NACK implosion* would occur if all receivers that missed the data replied with a NACK. In order to prevent NACK implosions diverse NACK suppression techniques are used.

Most NACK suppression is based on the use of random timers. All participants that want to transmit a NACK set a random timer. If the timer expires, a NACK is transmitted. If a NACK from another participant is received while the random timer is running the timer is reset. In this way duplicate NACKs are suppressed. The timeout value of the random timer may be equally distributed in an interval that depends on the round-trip-time between the sender of the data and the recipient that wants to send a NACK. This approach is used by the well-known scalable reliable multicast protocol SRM [3]. Other authors suggest distributing the timer exponentially over a fixed interval [15]. The latter is currently considered to be the more general, more efficient, and the more stable approach [7,4].

Typically all participants that have successfully received the requested packet are able to help with the repair of the packet. In order to prevent a repair implosion, the repair information is transmitted similar to NACKs by using a random-timer-based implosion avoidance mechanism.

When used for a static group of receivers with a single sender, approaches based on unstructured receiver groups are somewhat less scalable and efficient than those based on structured receiver groups. The reason for this is that there is no concept of local repair in approaches that use an unstructured receiver group. Repair packets are always transmitted to the whole group of recipients. Furthermore these mechanisms are slower in the repair of lost packets since they typically involve a random timer for NACK and repair implosion avoidance. On the other hand they are significantly less complex and do not incur the overhead of managing the structure formed by the recipients. They therefore support dynamic receiver groups as well as groups with many senders.

3.3. Forward Error Correction

The *forward error correction (FEC)* mechanism adds redundancy to the transmitted data in order to protect it from packet loss. It can be used for reliable multicast in two major ways. It either improves one of the two approaches described above or it can be a stand-alone approach to realize reliability for the transfer of bulk-data.

As a complementary mechanism forward error correction can be either proactive or reactive. *Proactive* FEC is used to protect the original data as it is being transmitted [17]. It does not rely on receiver feedback. Using proactive FEC can be very beneficial since in a multicast group it is very likely that at least one recipient will not receive a transmitted packet. If the packets are encoded with redundancy in an appropriate way, it is possible for receivers to miss distinct packets while still being able to completely recover the original data without requesting a retransmission. Furthermore proactive FEC supports the dissemination of time critical data, since request/reply round trips (and the delay introduced by random timers in approaches with an unstructured receiver group) for lost packets are reduced.

Reactive forward error correction is used to repair packet loss on demand. In reactive forward error correction the data is transmitted in rounds. After each round the receivers supply information about how many additional redundancy packets are required to repair packet loss. This can dramatically reduce the number of total repair packets that are required since a single redundancy packet can repair distinct packets that have not arrived at various receivers [14].

When FEC is used as a stand-alone mechanism to achieve reliability, the sender encodes the original data and redundancy information in a way that allows receivers to decode the original data once they have received a fixed number of arbitrary but distinct packets. Typically the number of distinct packets required to decode the data is equal to or slightly higher than the number of packets in the original data. The encoded packets are always sent repeatedly to a multicast group. A receiver joins the session at any time, receives the encoded packets, and may leave the session once it has received enough packets to decode the original data. The sender continues to send as long as at least one recipient is present in the session. Examples for multicast protocols that use this mechanism are the Digital Fountain approach [1], and the Asynchronous Layered Coding (ALC) reliable multicast protocol [11].

The usage of FEC as a stand-alone mechanism has the advantage that recipients may join the transmission at arbitrary times. Furthermore only rarely packets will be received that contain no useful data. This is not true for the other mechanisms where applications will frequently be burdened with repair packets that they do not need. Finally stand-alone FEC is highly scalable since it does not require any feedback from the receivers (besides the knowledge that at least one receiver is present). A drawback is that the data must consist of a single large chunk that is known in advance. Because of these characteristics the stand-alone FEC approach is typically used for bulk data transfer and not for real-time data.

4. RELIABLE MULTICAST FOR DISTRIBUTED INTERACTIVE MEDIA

From the media model it can be derived that there exist two distinct classes of information that need to be transmitted for a distributed interactive medium. On the one hand there is the data that is transmitted in real-time during the live session. This data comprises event and state information. On the other hand there is the environment information which is required by a participant before being able to participate in a session. In the following we examine which reliable multicast mechanisms are appropriate for these two classes of information.

4.1. Environment Information

The distribution of environment information has a number of important characteristics:

- Environment information is static and may be very large. For example, the postscript slides of a whiteboard presentation may have a size of several megabytes.
- Typically there will be a single sender for the environment information.
- The number of receivers may vary greatly, ranging from a few (less than 10) to thousands, depending on the medium.
- A receiving application will join the transmission and leave after it has received all relevant environment information.
- It is unlikely that the recipients of environment information will join the session simultaneously. The information should therefore be transmitted so that it is possible to use the data of an ongoing transmission even when applications missed the beginning of the transmission.
- There are no real-time constraints in the dissemination of environment information.
- The sender of the environment information does not need any knowledge about the recipients.

These characteristics show that the distribution of environment information is easy to handle. In particular it does not require real-time delivery and there exists only a single sender. Therefore all reliable multicast mechanisms described above could be used. However, since recipients will join the transmission of information at different times, stand-alone FEC-based mechanisms have an advantage when compared to the other methods: They efficiently support recipients that join during an ongoing environment transmission. We therefore view these mechanisms as an optimal fit for the distribution of environment information in distributed interactive media applications.

4.2. State and Event Information

State and event information is exchanged between the participants of a live session. They share several common traits:

- They require many-to-many communication. Usually all participants are able to send and receive this kind of information.
- The communication group is dynamic since participants may join and leave the session at arbitrary times.
- The number of participants can be moderate to high. For example, in a shared whiteboard session up to a few hundred users may participate while in networked computer games a thousand or more players may attend. However, it is quite unlikely that a single session will comprise more than a few thousand users. In the event that more participants are present it is likely that the communication will be split into multiple sessions [10].
- The size of the transmitted information is much smaller than the environment information. While events typically fit into a single network layer packet, states may occupy several network layer packets. However, the size of both events and states is one or more orders of magnitude smaller than the environment information.
- The data must be delivered under real-time constraints. These constraints may be rather loose or very strict, depending on the medium and whether event or state information is transmitted.
- Guaranteed reliability is not required. When an event or state transmission cannot be recovered the application is able to repair the problem at the application layer, e.g., by getting the correct shared state from a peer application. Such a functionality is part of any distributed interactive media application since it is required to realize important functionality such as consistency and support of latecomers.
- Packet loss for one sub-component and the repair thereof should not impair the data delivery for other sub-components. For example, when an event for a shared whiteboard page got lost, events for other shared whiteboard pages should not be buffered by the recipient until the packet loss has been repaired.

From these characteristics it can be derived that the dissemination of state and event information during a live session is quite demanding. Reliable multicast approaches that are based on a structured receiver group are not appropriate since a large number of senders exists and since the group membership is dynamic. The overhead for managing the receiver structure is not acceptable under these conditions. Reliable multicast solutions that rely exclusively on FEC are not appropriate since the data is fairly small, is not known in advance, and needs to obey real-time constraints.

This leaves us with approaches that use an unstructured group of receivers. These approaches seem quite adequate, since they can deal with many senders, dynamic groups, and small amounts of

data. In addition it has been shown that these approaches do scale up to the required number of a few thousand participants [3]. The main critical issue is that these approaches might not be able to fulfill the real-time requirement. However, this problem can be solved by using proactive FEC [17].

In order to be able to see how proactive FEC should be used, it is important to have a closer look at the differences between the transmission of event and state information. Typically the properties of an event transmission are as follows:

- An event can be encoded using only a small number of bytes, generally less than 30-50 bytes. It therefore fits into a single network-layer packet.
- An event needs to be delivered in time. An event delivered after it should have taken effect is a potential source of inconsistency and therefore usually triggers some sort of state resynchronization. Discrete media often are more tolerant of delayed events than continuous media. However, even discrete media suffer from delayed events when the ordering of events is destroyed by the (unexpected) delay.

State transmissions have somewhat different properties:

- The encoding of a state typically requires more bytes. The size of an encoded state can range from around 100 bytes for players in battlefield simulations to a few thousands of bytes for complex objects in a shared workspace.
- A state is extracted from the model at a certain point in time. The recipient of a state can decode a received state and extrapolate it to accommodate for transmission delay. Usually the maximum time during which the state of a sub-component can be extrapolated is the time between the extraction of the state information and the next event for that sub-component. For example, the state of a shared whiteboard page will remain valid until any one of the users interacts with the page. The recipient of state information will be able to extrapolate the state of the sub-component as long as no event has taken effect between the extraction and the extrapolation of the state. Some applications might even be able to automatically incorporate events when the state of a sub-component is extrapolated, thereby extending the time interval during which a transmitted state remains useful.

From the different characteristics of state and event transmission it can be derived that an application will likely profit from differentiated usage of proactive FEC. For events it might be a smart thing to use a high amount of proactive redundancy, since they are small and need to be received in real-time. State transmissions, on the other hand, should employ a much lower amount of redundancy since states can be large and do have a deadline which is not as strict as that of events.

Moreover, an application may want to react differently to the loss of event information than it does when state information is lost. For example, if an event gets lost it may make sense not to request a retransmission since the event would not be retransmitted in time to meet its deadline. The application may therefore wish to be informed about a lost event rather than require the reliable multicast service to repair the loss. On the other hand, for state transmission it is likely that the application would desire the repair of lost packets since it may be useful for a much longer time than an event.

In concluding it can be said that the transmission of state and event information is likely to be best served with a reliable multicast service that uses an unstructured receiver group in combination with proactive FEC. In addition the reliable multicast service may employ reactive FEC to repair lost packets efficiently. The interface of the reliable multicast service should allow the application to specify differentiated settings for the proactive FEC, and it should allow the application to specify what to do in the case that information for a specific message type gets lost.

5. CONGESTION CONTROL FOR DISTRIBUTED INTERACTIVE MEDIA

One important topic of current research in the area of reliable multicast transport protocols is congestion control [18,8]. Congestion control is required in order to adapt the data-rate of the transmitted information to the current status of the network. If congestion is ignored the network could become saturated, resulting in very high packet loss rates. In addition a reliable transport protocol without congestion control would treat transport protocols that do use congestion control in an unfair manner. In a congestion situation it would maintain a constant data rate while the transport protocols with congestion control mechanisms would lower their rate. It can therefore be expected that future reliable multicast transport protocols will offer some

sort of congestion control [7]. In the context of distributed interactive media it is of particular interest to understand when and how these congestion control mechanisms should be employed.

5.1. Environment Information

Since the transmission of environment information does not have to obey real-time constraints, the use of congestion control mechanisms is appropriate at any time. There are no negative side effects besides an increase in the transmission time (which is fair, considered that the network is congested).

5.2. Event and State Information

Events and states are transmitted in real-time during a live session. It is therefore not always appropriate to let a congestion control mechanism delay the transmission of this information. A delayed event may cause an inconsistent overall state of the medium which in turn might trigger a state repair process that burdens the (already) congested network with additional data. Similarly, a state which was buffered too long by a congestion control mechanism might become invalid.

It is therefore important that the application is able to signal how long an event or state may be delayed by a congestion control mechanism. If a congestion control mechanism identifies a situation where it is no longer possible to meet these deadlines, it should signal this to the application. The application in turn must then refrain from transmitting events and states. In particular the application should not try to repair inconsistencies during a congestion period, since this would increase the network load. Instead the application should wait until the network is no longer congested and then initiate a state repair to account for the event and state data that was not transmitted during the congestion period.

6. API DESIGN CONSIDERATIONS

Derived from the different characteristics of the dissemination of environment information on the one hand, and the live transmission of states and events on the other hand, we expect that two distinct reliable multicast services will be used for a distributed interactive medium. In the following we take a closer look on how appropriate application programming interfaces should look like in both cases.

6.1. API for Environment Transmission

As indicated in the discussion above we assume that a FEC only approach is used to disseminate the environment data. What an appropriate API for the transmission and reception of environment information could look like is depicted in Figure 1. The interface offered by the reliable multicast service for the sender is comprised of four methods: `setData` is used to tell the reliable multicast service what bulk data is to be transmitted. The bulk data is typically contained in a file. Upon receiving this message the reliable multicast service calculates the appropriate encoding of the data. As soon as a recipient joins the session, the reliable multicast service of the sender will start sending encoded packets in an appropriate way. When the last recipient leaves the session the service will stop sending packets until new recipients join the session. We expect that the information whether at least one receiver is listening to the transmission is provided by the reliable multicast transport protocol.

The application of the sender is informed about the status of the service with two methods `transmissionActive` and `transmissionInactive`. The first method is called when the service starts sending data while the second signals that the service has stopped sending data. If the data is no longer valid (e.g., because a session is finished) it is invalidated using `invalidateData`.

```
Sender's Reliable Multicast Service:

void setData(BulkData data)
void invalidateData()

Sender's Application:

void transmissionActive()
void transmissionInactive()

Receiver's Reliable Multicast Service:

void getData()

Receiver's Application:

void dataReceived(BulkData data)
```

Figure 1: API for Environment Transmission

The application of the recipient uses `getData` to signal that the service should retrieve the environment information from a specific multicast address. The service will join the address for the time required to receive the data and will then notify the application with `dataReceived`. The information about the number of packets required for a complete reception of the environment information should be signalled in-band by the reliable multicast transport protocol.

6.2. API for Event and State Transmission

As discussed above it can be expected that an approach with an unstructured receiver group, supplemented by proactive and, possibly, reactive FEC is best suited for the transmission of event and state information. There exist two opposing opinions on how an API for the reliable multicast based transmission of data like events and states should be realized. The first is typically voiced by developers of complex distributed interactive media. They demand an API that makes the reliable multicast transmission transparent to the application. They do not want to worry about how reliability is actually realized. For this group an interface similar to TCP sockets would seem optimal.

The second opinion is based on the concepts of application level framing (ALF) and integrated layer processing (ILP) [2]. Supporters of this opinion point out that a (sender) ordered, and reliable delivery of data may incur inefficiencies when packet loss occurs. This inefficiency stems from the fact that a packet loss may cause already received data to be buffered by a receiver until the packet loss has been repaired. For example, if the same sender transmits events for two different sub-components and the first event gets lost, a transparent, source ordered reliable multicast service would buffer the second event until the loss of the first event has been repaired. This is undesirable since it could cause the second event to miss its deadline, even though it could have been delivered in time. ALF-based interfaces to reliable multicast services seek to avoid this inefficiency by identifying the minimal units of data that the application can make use of. These minimal units of data are called application data units (ADUs). They are framed in a way so that a receiving application knows what to do with the data. ADUs are handed to the application as soon as they are received. Ordering between ADUs has to be performed by the application if such an ordering is desired. The application is also responsible for explicitly requesting lost ADUs and for providing the data for the repair of lost ADUs. Typically, a reliable multicast service with an ALF-based API does not buffer any ADUs. It simply transmits them in an appropriate way and performs loss detection. An application that uses a reliable multicast service with an ALF-based API is therefore likely to be more efficient at the cost of increased complexity at the application level. A prime example of such a reliable multicast service is SRM [3].

Application level framing fits the transmission of event and state data very well. The reason for this is that events and states essentially represent ADUs. Moreover in the requirements mentioned above it was mandated that event and state data for one sub-component should be delivered independent of the data for other sub-components. This cannot be realized with a transparent, source-ordered reliability interface. At the same time, however, the application should not be burdened with storing outdated ADUs, as it is required, e.g., by SRM. Ideally the application should be involved in the recovery of ADUs only when it chooses to be so. This requires a different API than commonly used for reliable multicast services based on ALF.

As a compromise we propose to use a subscription-oriented interface for the reliable multicast service. It offers the required flexibility and efficiency while the application is free to delegate repair functionality to the reliable multicast service. The interface uses the concept of ADUs. Figure 2 shows that an ADU contains the following information: an ID of the original sender of the ADU, an ID of the sub-component the ADU refers to, the data type (event vs. state data), a sequence number, and the actual data. The sender ID and the sub-component ID are unique and persistent for a given session. The sender ID may be obtained from a centralized source, by means of a distributed allocation algorithm or it may be calculated from local information (e.g., from an Ethernet card's MAC address). The unique sub-component ID can then be derived from the sender ID of the participant that introduced the sub-component into the session. A session participant may learn about the mapping from sub-component IDs to names that are meaningful to the application or the user either through dedicated protocols, such as SNAP [16], or by regular announcements as in RTP/I [13]. With the information contained in the ADU a receiving application knows exactly what the ADU refers to and what to do with it. Most importantly a receiving application can process each ADU independently.

A reliable multicast service must provide a `transmitADU` method. As parameters this method accepts the ADU and additional information about how to transmit it. The first part of this additional information identifies the amount of redundancy data that should be applied to the ADU as proactive FEC. Allowing the application to influence how proactive FEC is used is very

important since the application has important knowledge about the transmission requirements for a given ADU. The second part of the additional information identifies the time during which the ADU, including all redundancy, should be transmitted. For example, when transmitting an event the value for the time should be chosen so that all data does have the opportunity to arrive at the receivers before the deadline of the event is reached. This supports the use of congestion control mechanisms that are aware of the timing restrictions in the application. If the reliable multicast service is not able to meet these deadlines because of network congestion, it should notify the application by calling `congestion`. As mandated above, the application should then refrain from sending events and states until `congestionResolved` is called.

```
Application Data Unit (ADU) Definition:

Adu ::= SenderID SubID Type SeqNo Data

with Type element of {EVENT, STATE}

Reliable Multicast Service:

void transmitAdu(Adu adu,
                 float proactiveFec,
                 long deadline)

void setInterest(SubID subID,
                 Type type,
                 Qos qos)

with Qos element of {NONE, DETECT,
                     RELIABLE, ORDERED}

Application:

void receiveAdu(Adu adu)

void congestion()

void congestionResolved()

void aduLost(SenderID senderID,
             SubID subID,
             Type type,
             SeqNo seqNo)

void couldNotRecover(SenderID senderID,
                     SubID subID,
                     Type type,
                     SeqNo seqNo)
```

Figure 2: API for State and Event Transmission

Once an ADU has been handed to the reliable multicast service the application must be free to discard it. It is the reliable multicast services' job to buffer ADUs for retransmissions. This is appropriate since the reliable multicast service has information about the time for which the ADU needs to be buffered in order to make a reliable delivery very likely.

The `setInterest` method allows potential receivers to specify reliability quality-of-service settings on the level of sub-components and ADU types (event vs. state ADUs). The parameters allow to specify a pair of one sub-component and one ADU type, and a quality-of-service that should be used by the reliable multicast service for ADUs that belong to the pair. We envision that the following quality-of-service settings are of use for distributed interactive media:

- NONE. ADUs are delivered as they are received. If an ADU gets lost it will not be recovered, nor will the application be informed about the loss. This is the setting that should be chosen for sub-components that are not of interest for the local recipient.
- DETECT. Same as above, but the application is informed if an ADU gets lost. This requires sequence number monitoring and tail loss detection. Loss detection may be very useful for events in continuous distributed interactive media. A retransmission of the lost event may be useless for a continuous medium since the deadline of the event would not be kept. Instead of requesting a retransmission, the application may therefore trigger the repair of the damaged shared state.
- RELIABLE. ADUs are delivered immediately after they have been received. If an ADU or a part of an ADU gets lost, the loss will be repaired by retransmission. This is frequently required for state transmissions.
- ORDERED. ADUs transmitted by a single participant are handed to the application in the order in which they were transmitted. If necessary, ADUs are buffered by the receiver until the packet loss for previous ADUs has been repaired. Only lost ADUs of the same type, referring to the same sub-component, may cause an ADU to be buffered. This may be used by non-continuous media for event and state transmissions.

The application must provide a `receiveADU` method which is invoked by the reliable multicast service to deliver an ADU to the application. In the event that the application has chosen to use the loss detection quality-of-service, it receives an `aduLost` message whenever an ADU gets lost

for the specified sub-component/ADU-type pair. This function contains the ID of the lost ADU's sender, the sub-component ID, the type, and the sequence number.

The final function that needs to be provided by the application is required because ADUs should be buffered for retransmission only during a limited time. This amount of time should be large enough so that it is very unlikely that a repair request will arrive after the ADU has been discarded. However, since a reply to the retransmission request cannot be guaranteed with absolute certainty, the application needs to be informed should this situation arise. This is done using `couldNotRecover`. In this case the application needs to repair the problem. An application for a distributed interactive medium is able to perform this repair since the same functionality is also needed in other situations, e.g., to accommodate latecomers. It should be noted that this is an exceptional situation which should occur only on very rare occasions.

7. CONCLUSION AND OUTLOOK

In this paper we investigated the use of reliable multicast for distributed interactive media, such as shared whiteboards, networked computer games and distributed virtual environments. In a first step we presented our model for the distributed interactive media class and summarized existing mechanisms to achieve reliable multicast. With this information we were able to reason what reliable multicast mechanisms should be used for which aspects of distributed interactive media. In particular we found that the static and large environment information is best transmitted using a FEC-only approach. The main reason for this choice is the ability to join an ongoing transmission at any time. For the data transmission during a live session (i.e., events, and states) only mechanisms with an unstructured receiver group are acceptable. In order to support the real-time nature of event data these mechanisms need to be complemented by proactive FEC.

It was discussed how to design an appropriate application programming interface so that convenient access to the reliable multicast service is guaranteed for distributed interactive media applications. We proposed to use a compromise between traditional interfaces where reliability is established transparently for the application, and those approaches which rely on application level framing in combination with applications that are network aware. In particular our design prevents the typical inefficiencies of a completely transparent approach, while it does not burden the application with tasks like the buffering of data for retransmission purposes.

The work presented here has shown that the most important pieces of functionality required to provide an appropriate reliable multicast service for distributed interactive media are already available. What remains to be done is to take the existing mechanisms, combine them in the right way, and provide them with an appropriate interface such that applications for distributed interactive media can make efficient and easy use of the resulting reliable multicast service. We are currently investigating diverse reliable multicast implementations in order to adapt them to the findings presented here.

8. ACKNOWLEDGMENTS

This work was inspired by a number of discussions within the European Commission Telematics for Research Project RE4007 (MECCANO). In particular we wish to thank Colin Perkins for a very helpful discussion about ALF and reliable multicast. We also thank the anonymous reviewers for providing knowledgeable and detailed comments. This work was partially supported by the DFG (Deutsche Forschungsgemeinschaft) within the V3D2 Digital LibraryInitiative.

9. REFERENCES

1. J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. *In: Proc. of the ACM SIGCOMM '98 conference on Applications, Technologies, Architectures, and Protocols for computer communication*, Vancouver, Canada, September 1998, pp. 56 - 67.
2. D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In: *Proc. of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM) '90*, 1990, pp. 201 - 208.
3. 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.

4. T. Fuhrmann. *On the Scaling of Feedback Algorithms for Very Large Multicast Groups*, Technical Report TR-01-2000, Reihe Informatik, Department for Mathematics and Computer Science, University of Mannheim, February 2000.
5. J. J. Garcia-Luna-Aceves, and B. Levine. End-to-End Reliable Mulitcast. Chapter 6 of *Multimedia Communications*, F. Kuo, W. Effelsberg, J. J. Garcia-Luna-Aceves (Eds.), Prentice Hall, Upper Saddle River, 1998.
6. 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.
7. M. Handley, B. Whetten, R. Kermode, S. Floyd, and L. Vicisano. *The Reliable Multicast Design Space for Bulk Data Transfer*. Internet-Draft: draft-ietf-rmt-design-space-00.txt. June 1999. Work in progress.
8. M. Kadansky, D. Chiu, J. Wesley, J. Provino. *Tree-based Reliable Multicast (TRAM)*. Internet-Draft: draft-kadansky-tram-02.txt. January 2000. Work in progress.
9. C. Kuhmünch, T. Fuhrmann and G. Schöppe. Java Teachware - The Java Remote Control Tool and its Applications. In: *Proc. of ED-MEDIA/ED-TELECOM'98*, Freiburg, Germany, 1998, available on CD-ROM.
10. E. Lety and T. Turletti. Issues in Designing a Communication Architecture for Large-Scale Virtual Environments, In: *Proc. of NGC'99*, Pisa, Italy, Springer LNCS, Vol. 1736, 1999, pp. 54-71.
11. M. Luby, J. Gemmell, L. Vicisano, L. Rizzo, J. Crwocroft, and B. Lueckenhoff. *Asynchronous Layered Coding*. Internet-Draft: draft-ietf-rmt-pi-alc-00.txt. March 2000. Work in progress.
12. 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, January 1999, pp. 240 - 251.
13. 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.
14. J. Nonnemacher, and E. Biersack. Parity Based Loss Recovery for Reliable Multicast Transmission. In: *IEEE/ACM Transactions on Networking*, Vol. 6, No. 4, 1999, pp. 349- 361.
15. J. Nonnenmacher, and E. Biersack. Scalable Feeback for Large Groups. In: *IEEE/ACM Transactions on Networking*, Vol. 7, No. 3, 1999, pp. 375- 386.
16. S. Raman and S. McCanne. Scalable data naming for application level framing in reliable multicast. In: *Proc. of 6th ACM international conference on Multimedia*, Bristol, UK, 1998, pp. 391 - 400.
17. D. Rubenstein, J. Kurose, and D. Towsley. Real-Time Reliable Multicast Using Proactive Forward Error Correction. In: *Proc. of IEEE NOSSDAV'98*, Cambridge, UK, July 1998.
18. B. Whetten, M. Basavaiah, S. Paul, T. Montgomery, N. Rastogi, J. Conlan, and T. Yeh. *The RMTP-II Protocol*. Internet-Draft: draft-whetten-rmtp-ii-00.txt. April 1998. Work in progress.