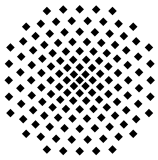

Universität Stuttgart
Fakultät Informatik

Constructing Height-Balanced Multicast Acknowledgment Trees with the Token Repository Service

Authors:
Dipl.-Inf. C. Maihöfer
Prof. Dr. K. Rothermel

Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

Constructing Height-Balanced Multicast Acknowledgment Trees with the Token Repository Service

C. Maihöfer, K. Rothermel

Bericht 1999/15
November 1999

Constructing Height-Balanced Multicast Acknowledgment Trees with the Token Repository Service

Christian Maihöfer and Kurt Rothermel
Institute of Parallel and Distributed High-Performance Systems (IPVR),
University of Stuttgart, Germany
{maihoefer,rothermel}@informatik.uni-stuttgart.de

***Abstract* -- Many reliable multicast protocols use so-called ACK-trees to avoid the well-known acknowledgment implosion problem in case of large multicast groups. For constructing ACK trees, usually expanding ring search techniques are applied. Our simulation results show that those techniques have scalability problems itself.**

In this paper, we propose a novel approach for building ACK trees, the token repository service (TRS). The basic concept of our approach is a token, which represents the right to connect to a certain node in the corresponding ACK tree. For each node in the ACK tree TRS stores a token for each potential successor this node can accept. A node that wants to join a group requests TRS for an appropriate token. The TRS protocol described in this paper provides for height-balanced ACK trees.

Our simulation results show that the created height-balanced ACK trees have significant benefits. They reduce round trip delay and optimize reliability in case of node failures. Moreover, compared to expanding ring search, TRS results in a much lower message overhead.

I. Introduction

A great variety of todays networked applications require reliable one-to-many or many-to-many communication. For those applications a number of reliable multicast protocols [1,2,3,4,5,6] have been proposed. Most of them are based on IP multicast [7,8] and use positive or negative acknowledgments (ACKs) to confirm delivery. In large multicast groups, the sender may be overwhelmed by the amount of acknowledgments received, which is the well-known ACK implosion problem. The most promising approach to address this problem is to organize the group of receivers in a so-called ACK tree [3,4,5,6]. While the multicast messages are sent directly to the receivers (e.g., by using IP multicast), the responded ACK messages are aggregated and propagated along the edges of the ACK tree.

When a new member wants to join the multicast group, it has to be connected to the group's ACK tree. Several techniques have been proposed in the literature for maintaining the ACK tree, most of them are based on expanding ring search (ERS) [9]**.** The advantage of ERS is its simplicity and fault tolerance. On the other hand, our simulations results will reveal several drawbacks of ERS, like poor scalability and problems correlated with the various multicast routing protocols.

In this paper, we propose the *token repository service* (TRS) as an alternative approach for constructing ACK trees. It is based on the concept of a distributed token repository, where a token represents the right to connect to a certain node in an existing ACK tree. A new member willing to join the group's ACK tree asks the TRS for a token of that group. TRS selects and delivers a token, which identifies the node in the group's ACK tree the new member can connect to. For each successor the new member can accept, a token is generated and stored by TRS.

It is important to mention that TRS achieves the same level of fault tolerance as ERS. Even if all token information becomes unavailable due to node and communication failures TRS is still operational.

In [10] we have already proposed TRS with the random-choice strategy (TRS-RC). Simulations have shown that in terms of message overhead and reliability of the created ACK trees, TRS-RC is a substantial improvement over ERS. However, in terms of round trip delay TRS-RC performs only with core based routing approaches better than ERS. Moreover, the created ACK trees are not completely height-balanced, which means that they are not optimal in terms of reliability.

The scheme presented in this paper, TRS with minimal-height strategy (TRS-MH), allows to create height-balanced ACK trees. A configurable deviation parameter specifies the maximum acceptable height deviation from strictly height-balanced trees. A deviation of zero results in strictly height-balanced trees, which leads to significantly lower round trip delays as compared to TRS-RC and ERS. On the other hand, creating strictly height-balanced ACK trees causes more message overhead than constructing less balanced trees. Therefore, the deviation parameter determines the trade-off between message overhead and delay/reliability. The effects of this parameter will be illustrated by our simulation results.

The remainder of this paper is structured as follows. The next section discusses related work and provides for

the necessary background. In Section III, the TRS approach based on the minimal-height strategy will be described in detail. In Section IV, we compare TRS with the ERS based approaches in terms of performance. Finally, we conclude with a brief summary.

II. Problem Statement and Related Work

As stated above, the majority of reliable multicast protocols use the concept of ACK trees to avoid the well-known implosion problem. For each multicast group there exists an ACK tree, whose nodes are the members of the multicast group. ACK messages flow in a leaf-to-root direction, where an ACK sent by a node acknowledges receipt for the entire subhierarchy of this node. To avoid the implosion problem, each node in the ACK tree is assumed to have an upper bound on the number of its successors. We will call a node *k-bounded* if the number of successors never exceeds *k*. A k-bounded node is defined to be occupied if it has *k* successors. The bound chosen for a node depends on various characteristics, such as the node's reliability, load, and performance.

When a new member joins the group, it must be connected to a non-occupied node in the corresponding ACK tree. In order to allow for large multicast groups, the underlying join mechanism must be scalable and efficient. Moreover, it is desirable that this mechanism constructs well-formed ACK trees, i.e. those with minimal height.

The height of the ACK tree affects both delay and reliability. Low delays are desirable since the average throughput of a reliable (multicast) channel based on the PAR scheme [11] is limited by the quotient of buffer size / round trip delay. The lower the delay, the higher is the throughput respectively the lower is the required buffer size.

The multicast service may be disrupted for a node if one of its predecessors in the ACK tree becomes unavailable. Therefore, the lower the number of predecessors the higher the reliability from this node's perspective. So, the average path length of the ACK tree can be taken as a quality criteria for reliability.

Most approaches to establish ACK trees are variations of ERS, which was first proposed in [9]. When a node wants to join the ACK tree, basic ERS [3] simply multicasts a join request to the members of this group. In order to decrease network load and to find a predecessor as close as possible to the searching node, the search is limited by a search scope. The initial request is sent with an time-to-live (TTL) of 1 and thus is limited to the sender's LAN. When a non-occupied node in the group's ACK tree receives this message, it returns an answer. If no node answers within a certain time, the requestor multicasts the join request with an increased TTL. It repeats this process until an answer arrives or the maximum TTL of 255 is reached. The node that answers first becomes the predecessor of the new member.

In a variation of ERS, called expanding ring advertisement (ERA), the nodes that are already in the ACK tree actively search for successors [4,6]. Non-occupied ACK tree nodes send multicast invitation messages to gain further successor nodes. Since the receivers need not send multicast messages to join the ACK tree, no support for bidirectional multicast is required. Some protocols use a combination of ERS and ERA (e.g., see [5]).

The great advantage of ERS and ERA is its implicit fault tolerance. On the other hand, our simulation results in Section IV will show that ERS as well as ERA may cause a huge message overhead. Another shortcoming of ERS is its dependency on the various routing protocols, each has its own problems with respect to ERS. ERS with DVMRP [12] routing leads to a vast overhead at all involved routers because a new multicast routing tree is to be build for each node joining a group. For pure receivers, these trees are only used for running ERS. With shared tree approaches like PIM-SM [13], the use of ERS results in a traffic concentration at the core, an even higher message overhead, and ACK trees of poor quality. A serious drawback of ERA is the message overhead due to the invitation messages, which are sent even if no node wants to join.

In [10], we have proposed the basic TRS with random-choice strategy (TRS-RC) for constructing multicast ACK trees. We have shown that TRS-RC causes a much lower message overhead than the ERS or ERA schemes. Moreover, in terms of reliability and round trip delay TRS-RC performs in many cases better than ERS and ERA (see Section IV for details).

TRS with minimal-height strategy (TRS-MH), presented in this paper, constructs height-balanced ACK trees. Simulation results in Section IV show that in terms of delay and reliability, TRS-MH is superior to TRS-RC.

III. The Token Repository Service with Minimal-Height Strategy (TRS-MH)

A. Basic Principle and Interface

The core concept of our approach are tokens, where a token represents the right to connect to a particular node in a given ACK tree. When a k-bounded node joins the ACK tree, *k* tokens are generated and stored in the TRS. The joining node is called the tokens' owner. A token is therefore identified by its group and owner. We define the height of a token to be the height of its owner in the corresponding ACK tree.

When a node wants to join a given group, it asks the token repository service for a token belonging to this group. The repository service selects a token of this group and delivers it. The node receiving this token can then connect to the token's owner in the corresponding ACK tree. When a node leaves a group, it returns the token to the repository service, which then can be reused by some other node joining this group later. Since only *k* tokens are generated for a k-bounded node, no more than *k* successors can be connected to it in the corresponding ACK tree. Table 1 shows the operations provided by the TRS.

TABLE 1 Operations provided by the TRS

repCreateGroup (Group, AckRoot, K): This operation announces the group identified by *Group* to the TRS. The root of *Group*'s ACK tree is identified by *AckRoot*, which is *K*-bound.
repDeleteGroup (Group): This operation deletes all state information associated with *Group* in the TRS.
repJoinGroup(Group, NewMember, K) ret. Token This operation is called when the node identified by *NewMember* wants to join *Group*, where *NewMember* is *K*-bounded. The operation returns a token that identifies the node, *NewMember* is supposed to connect to.
repLeaveGroup(Group, Member): This operation deletes all tokens owned by *Member* for *Group* in the TRS.
repAddToken(Group, Owner): This operation is called when a successor disconnects from *Owner* in *Group*'s ACK tree. This operation adds a new token owned by *Owner* into the TRS.
repRemoveToken(Group, Owner): This operation is called when a node identified by *Owner* in *Group*'s ACK tree has accepted a new successor via ERS. It removes one of *Owner*'s tokens.

B. Implementation as a Distributed Server Hierarchy

In order to achieve scalability, the TRS is implemented as a distributed system consisting of a hierarchy of token repository servers, or *repServers* for short. In our approach, the network is structured into hierarchical domains. Leaf domains encompasses a disjunct set of nodes, inner domains include all successor domains and finally the root domain includes all nodes of the network. We assume that domains group the network by communication distance, i.e. the communication distance between two nodes belonging to the same domain is typically smaller than between two nodes in different sibling domains. Figure 1 depicts the hierarchical domain structure. Each domain is administered by a repServer. For example repServer S_1 in Figure 1 is responsible for domain 1 consisting of nodes N_{1x} and repServer S_5 is responsible for domain 5, which consists of several subdomains.

A leaf repServer, responsible for a certain node in its domain, is called this node's *home repServer*. For example in Figure 1, S_1 is the home repServer for all nodes N_{1x} of domain 1. Nodes access the TRS only via their home repServer, which are always leafs in the repServer hierarchy. Non-leaf repServers are responsible for token searching (see Section III.F for details).

As mentioned before, the TRS stores tokens, representing the right to connect to a certain predecessor node in the ACK tree. These tokens are stored on leaf repServers in so-called *token baskets*. Each leaf repServer has a basket for each known group, which includes all tokens of the corresponding group.

When a leaf repServer is asked for a token of a given group and no suitable token is locally available for that group, it starts the token search procedure. A token is defined to be suitable if there exist no tokens of the same group with a lower height. To facilitate searching for each group a so-called group tree is maintained, which is a subtree of the hierarchy of repServers. A group's group tree contains all leaf repServers that store a token basket of this group and all ascendants of these nodes in the repServer hierarchy. repServers store group tree information in so-called group records. With TRS-MH, a group record stored at a repServer includes the minimal height of the tokens available in this repServer's subhierarchy for the corresponding group.

Let us briefly sketch the search procedure for TRS-MH. A repServer starts a token search by sending a search request to its predecessor. If the receiver of this request is not part of the group tree, it just forwards the request to its predecessor. Otherwise, it determines -

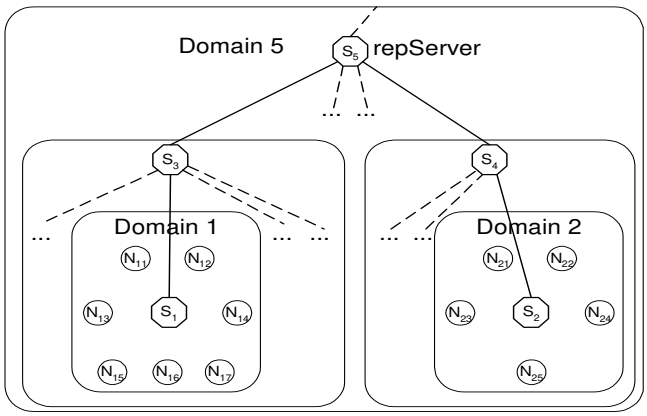

Figure 1: Domain structure

depending on the height information included in the local group record - whether its subhierarchy contains a suitable token. If no suitable token is available, the search domain is enlarged by forwarding the token request to its predecessor. Otherwise, the search request is forwarded in a root-to-leaf direction along the edges of the group tree until it reaches a leaf repServer storing a suitable token. Note that this search procedure together with our assumption that a node within the same domain is closer in terms of communication distance than a node in a sibling domain ensures the following:

(a) From all tokens available for a given group always one with the lowest height is selected, and

(b) if there are several tokens with the lowest height, one whose owner is as close as possible to the joining node is selected.

Note that the above mentioned deviation parameter can be used to trade-off between the communication distance between nodes in the ACK tree and the height of this tree. If, for example, the acceptable deviation is 2, then a leaf repServer may select a local token (i.e., whose owner is close) even if this token exceeds the minimal height of the globally available tokens by 2.

In contrast, TRS-RC [10] does not store height information in non-leaf group records. If the random choice strategy is applied, the token is forwarded in a leaf-to-root-direction until a repServer is reached whose subhierarchy holds *some* token of the corresponding group. Then the token request is forwarded to some leaf repServer in the subhierarchy holding such a token, which is chosen randomly if there are several of them. Consequently, with TRS-RC, the token is selected whose owner is as close as possible to the joining node, independent of this token's height.

C. Group State Information

The group record structure at leaf repServers is depicted in Table 2. Each group record contains a token basket, which is a set of token packets belonging to the same group. The token packet structure is given in table Table 4. Each token packet contains a number of tokens belonging to the same owner.

Besides the token basket, a leaf repServer's group record contains the fields *MinHeightGlobal* and *MinHeightLocal*. The first specifies the minimal height of all tokens in the entire repository hierarchy, while the latter records the minimal height of all locally available tokens. As we will see in Section III.F, *MinHeightGlobal* is only a lower bound of the group's minimal token height and not necessarily the exact value.

The group record structure for non-leaf repServers is given in Table 3. Besides *MinHeightGlobal*, vector *MinHeightSub* is included, which determines the group tree's successor nodes. *MinHeightSub*[*s*] denotes the vector's entry for successor *s*. If a successor *s* does not belong to the group's tree, *MinHeightSub*[*s*] contains a *null* entry. Otherwise, it specifies the minimal height of the group's tokens available in domain *s*. In the following, we will use min(*MinHeightSub*) to denote the minimal height of all entries in *MinHeightSub*.

TABLE 2 Leaf node group record

Attribute	Description
Group	Unique multicast group identifier
TokenBasket	Set of token packets
MinHeightGlobal	Minimal height of tokens in the entire repository hierarchy
MinHeightLocal	Minimal height of all local tokens belonging to *Group*

TABLE 3 Non-leaf node group record

Attribute	Description
Group	Unique multicast group identifier
MinHeightGlobal	Minimal height of tokens in the entire repository hierarchy
MinHeightSub	Vector determining the minimal token height for each successor domain
ExpDate	Date when the group record expires

TABLE 4 Token packet structure

Attribute	Description
Owner	The *Owner* of the tokens in this packet
Height	Height of *Owner* in the corresponding ACK tree
Tokens	Amount of tokens in this packet
ExpDate	Date when the packet expires

D. Group Record Updates

A group tree may grow and shrink during its lifetime, and the contents of group records may change. A group record at a leaf repServer is to be established when the first set of tokens associated with the group are created locally, and it is deleted when the last token has been removed from it. If a new group record is created at a leaf repServer, this leaf repServer is linked to the group tree

by sending its predecessor a *HeightUpdate* message, containing the group's identifier as well as the sender's *MinHeightLocal* and *MinHeightGlobal* values.

A non-leaf repServer receiving a *HeightUpdate* message checks whether the corresponding group record already exists. If it already exists, the receiver updates *MinHeightSub* and *MinHeightGlobal* accordingly. Otherwise, it creates a new group record for this group and sets the *MinHeightGlobal* and corresponding *MinHeightSub* entry to the *MinHeightGlobal* and *MinHeightLocal* value included in the message.

If a group record is removed, a *HeightUpdate* message is sent to the repServer's predecessor, indicating that no local token is available anymore. If a non-leaf repServer receives such a message, the corresponding *MinHeightSub* entry is set to *null*. If all entries are *null*, this repServer does no longer belong to the group tree, and hence its group record can be removed also, causing a *HeightUpdate* message to be sent to its predecessor.

If we want to built a strictly height-balanced ACK tree, each time a leaf repServer delivers a new token, whose height is greater than its local *MinHeightGlobal*, this value is updated to the height of this token. As we will see in *Section III.F* this can only happen after a token search has been performed. If we take into account the deviation parameter *Dev*, *MinHeightGlobal* is only updated to the token's height if a global token search has resulted in a token with height greater than *MinHeightGlobal + Dev.* Remember that *Dev* specifies the maximum allowed height deviation from a strictly height-balanced tree. With this updating strategy we make sure that *MinHeightGlobal* is always a lower bound of the minimal token height, which is needed to determine when a global token search is required.

Changes of group records in non-leaf repServers are propagated accordingly to predecessor nodes.

Now let us briefly describe the expiration date mechanism. All state information are maintained following the soft state principle [14]. Both, group records and token packets are associated with an expiration date. If an expiration date is not extended when it expires, the corresponding piece of information will be discarded automatically. Although our protocols allow for discarding state information explicitly, this mechanism is needed to ensure that all state information will be eventually removed even in the presence of node and communication failures. In addition, having such a mechanism in place allows to use light-weight protocols for explicitly deleting state information.

How can expiration dates be extended? Clearly, the lifetime of a token packet depends on the lifetime of its owner. When a token packet expires, the repServer storing this packet asks the packet's owner to extend the expiration date. If the owner responds, the expiration date is updated accordingly, otherwise the entire package is removed. If the token basket of the corresponding group becomes empty, the token basket's group record is deleted and the leaf repServer removed from the group tree.

When a group record expires, the repServer storing this record asks the successor repServers whether they are still part of the corresponding group tree. If at least one of them responds positively, then the record's expiration date is extended accordingly, otherwise it is deleted and removed from the group tree.

E. Creating and Deleting Groups

In this and the following section we describe the group management operation, i.e. create group, delete group, join group and leave group in detail. When describing the protocol we assume the absence of failures. We will consider communication and node failures in Section III.H.

When a new group is created, tokens must be created and an initial group tree must be established. Figure 2 illustrates a scenario, where a node creates a group and two other nodes join this group. Assume that S_1 is the home leaf repServer at which the *repCreateGroup(Group*[1]*, AckRoot, K)* operation of node N_{11} was issued. S_1 creates a group record with token basket for *Group* with the amount of tokens specified by *K,* where *AckRoot* becomes the owner of these tokens.

The group tree to be established consists of S_1 and all ancestors of S_1 in the repServer hierarchy. After creating the token basket, S_1 sends a *CreateGroup* request to its predecessor. On receiving a *CreateGroup* request, a repServer creates and initializes a group record and again forwards the request until the root repServer is reached. The *MinHeightSub* entry is initialized with 2, since all tokens in this domain are of height 2; all other entries are initialized with *null*, indicating that these successor domains do not belong to the group tree. *MinHeightGlobal* is initialized with 2 because the lowest height of the group's globally available tokens is 2, too.

When the operation *repDeleteGroup(Group)* is issued at a leaf repServer, this server deletes the *Group*'s group record with all token packets and sends a *DeleteGroup* request to its predecessor. Non-leaf repServers forward this request along the edges of *Group*'s group tree, and each repServer receiving this request deletes all state information associated with *Group*. Note that this explicit discarding of state information is only an optimization

[1]. Here we assume an external mechanism that ensures group ids to be unique. See [10] for details.

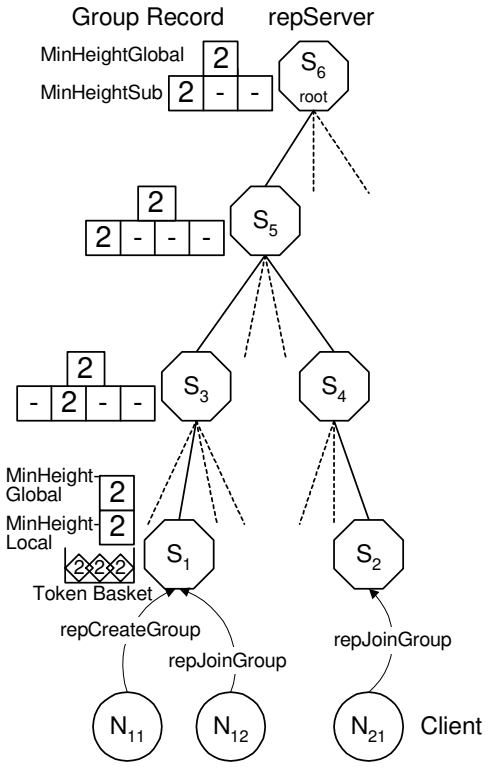

Figure 2: Group state information after creating a group

since the expiration date mechanism ensures that all state information is removed eventually.

F. Joining and Leaving Groups

When a *repJoinGroup*(*Group*, *NewMember*, *K*) *returns* (*Token*) operation is called, the called leaf repServer checks its group record to see, whether it has a suitable token in the *Group*'s token basket. A suitable token is available at a leaf repServer if the following condition is fulfilled:

$$MinHeightLocal \leq MinHeightGlobal + Dev,$$

where *Dev* is the deviation parameter mentioned above, i.e. strictly height-balanced trees are constructed if *Dev* is set to zero.

If a suitable token is locally available, a token with height *MinHeightLocal* is removed from the token basket and returned to the caller of *repJoinGroup*. The home repServer also generates a new token packet for *NewMember* with *K* tokens and puts it into *Group*'s token basket.

In the example depicted in Figure 2 and the following ones, we will assume *Dev*=0. Figure 2 shows the outcome of a *repJoinGroup* performed by node N_{12}. Its home repServer S_1's check results in *MinHeightGlobal* is equal to *MinHeightLocal*. It removes a local token with height 2, returns it to N_{12} and generates a new token packet with the owner N_{12}.

If a suitable token is not available at a leaf repServer, this server starts the search procedure. In the first phase, a *TokenRequest* request is forwarded in leaf-to-root direction until a repServer is found that is part of *Group*'s tree and the following condition is satisfied:

$$\min(MinHeightSub) \leq MinHeightGlobal + Dev.$$

If this condition holds at a repServer, a suitable token can be found in this repServer's subhierarchy, and hence the search domain needs no further enlargement.

Such a situation is depicted in Figure 2. N_{21} initiates a *repJoinGroup* operation at its home repServer S_2. S_2 has no token basket for the requested group. Therefore, it starts a global token search. S_5 is the first repServer that is part of the group tree. Since *MinHeightGlobal* is equal to min(*MinHeightSub*), the first search phase stops and the second, root-to-leaf directed search phase, is initiated by S_5.

In the second search phase, each repServer forwards the *TokenRequest* message to a subhierarchy *s* with *MinHeightSub*[*s*] = min(*MinHeightSub*) until a leaf repServer is reached. If more than one subhierarchy satisfies this condition, the repServer randomly selects one of them. Finally, the found leaf repServer removes a token with height *MinHeightLocal* from *Group*'s token basket and delivers it directly to the searching leaf repServer. In the example depicted in Figure 2, S_5 has to choose S_3, which itself choose leaf repServer S_1.

When receiving the token, the searching leaf repServer establishes a new group record with a token basket, including a token packet owned by *NewMember* and finally delivers the received token to the caller of *repJoinGroup*. In order to connect itself to *Group*'s group tree, it sends a *HeightUpdate* message to its predecessor (see Section III.D).

Figure 3 depicts the group state information after N_{21} has received the token. *MinHeightGlobal* has still the value 2 since S_1 has still tokens with this height. *MinHeightSub*[S_4] at S_5 and *MinHeightSub*[S_2] at S_4 are set to 3, the height of the tokens at S_2.

Subsequent *repJoinGroup* operations will first consume S_1's tokens with height 2. If the last token with height 2 has been consumed, the next *repJoinGroup* will result in a global token search, because *MinHeightGlobal* still indicates the existence of a token with height 2 at most group tree's repServers. Of course, the global token search cannot find such a token, hence the token with the lowest height of all remaining ones is chosen and *MinHeightGlobal* of the searching repServer is updated. The predecessor is informed of this update by a *HeightUpdate* message as already explained in Section III.D. Assume that in Figure 3, S_1 has delivered its last token with height 2. Then S_1 sends a *HeightUpdate* message to its predecessor with the new minimal token height of 3. This

message is forwarded to the ancestors until the root repServer S_6 receives it. Now a *repJoinGroup* is initiated at S_2 and hence S_2 starts a token search since *MinHeightGlobal* denotes that possibly a better token could be found globally. The search request is forwarded until S_6 is reached, where it also finds no successors with a suitable token of height 2. Finally S_2 chooses a local token. Figure 4 shows the group state information after the processed updates initiated by S_2. Note that all *MinHeightGlobal* entries on the path from S_2 to S_6 are updated to token height 3, but not those on other paths. For example S_1 and S_3 still have an *MinHeightGlobal* entry of 2, indicating that possibly a token with height 2 could be found, which obviously is not the case.

Now let us consider these exceptional cases in more detail. First of all, if the first search phase discovers no token with a smaller height than the searching repServer's *MinHeightLocal* value, then the searching leaf repServer chooses a local token. The search can be stopped if at a repServer's group record *MinHeightGlobal* is equal to *MinHeightLocal*.

If in our example depicted in Figure 4 S_1 initiates a token search as its *MinHeightGlobal* value indicates that a better token might be available, the search process can be terminated at S_5. Since *MinHeightGlobal* is 3, and there are also local tokens at S_1 with the same height, S_1 chooses a local token. Of course, S_1 initiates the update of *MinHeightGlobal* after delivering the local token.

If a search request arrives at the root repServer, the root checks if min(*MinHeightSub*) is smaller than the searching repServer's *MinHeightLocal* value included in the TokenRequest message. If this is the case, the root repServer sends the *TokenRequest* to the appropriate successor repServer. The search algorithm is summarized in Figure 5.

```
H receives repJoinGroup (Group, NewMember, K) from N:
    if MinHeightLocal > MinHeightGlobal + Dev
        send TokenRequest(Group, MinHeightLocal, H) to predecessor
    else
        send token with height == MinHeightLocal to N

R receives TokenRequest(Group, MHL, H) from successor:
    /* MHL is the MinHeightLocal of the searching leave repServer H */
    if (MHL <= MinHeightGlobal + Dev) or
        (R == root and MHL <= min(MinHeightSub))
        take local token at H
    elseif (min(MinHeightSub) <= MinHeightGlobal + Dev) or
        (R == root)
        send TokenRequest(Group, MHL, min(MinHeightSub), H)
            to a succ. S with MinHeightSub[S] == min(MinHeightSub)
    else
        send TokenRequest(Group, MHL, H) to predecessor

R receives TokenRequest(Group, MHL, Height, H) from a predecessor:
    /*  MHL is the MinHeightLocal of the searching leaf repServer H
        Height is the height of the searched token */
    if min(MinHeightSub) <= Height
        send TokenRequest(Group, MHL, Height, H) to a succ. S
            with MinHeightSub[S] == min(MinHeightSub)
    else
        send HeightUpdate and TokenRequest(Group, MHL, H) to pre.
```

Figure 5: Search algorithm

During the token search, inconsistencies in the search records may occur due to network partitioning and/or node crashes (see Section III.H.). Temporary inconsistencies can also occur due to the time needed to propagate updates.

Finally we describe the *repLeaveGroup* operation.

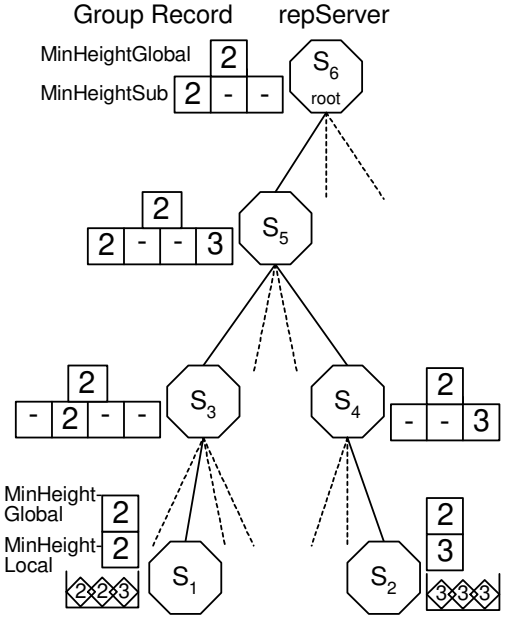

Figure 3: Group state information after a join operation at S_2

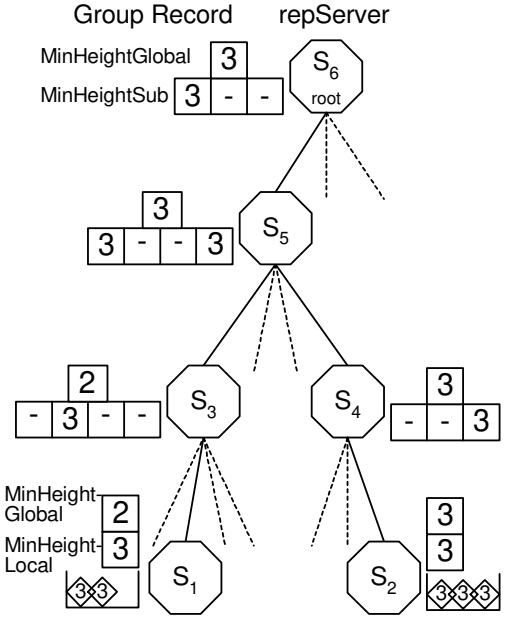

Figure 4: Group state information after S_1 has delivered the last token with height 2

When a node, say *N*, leaves a group, it conceptually releases a token owned by its predecessor. Hence, *N*'s predecessor is requested to add this token by means of the *repAddToken* operation to the token basket of its home repServer when it recognizes that *N* leaves the group. As we assume that a node has no descendants in the ACK tree when it leaves the group, all tokens owned by *N* should be in the group's token basket stored on *N*'s home repServer at the time the *repLeaveGroup* operation is called. When receiving this call, *N*'s repServer removes *N*'s token packet from the group's token basket. If the token basket becomes empty, it's group record is removed and a *HeightUpdate* message is sent to *N*'s predecessor.

G. Cashing of Group Information

In our descriptions so far, we have not considered that a repServer's memory is a limited resource. Since we cannot guarantee that a repServer can store all created tokens, we must introduce a token caching mechanism.

We assume that a repServer has a token cache, i.e. it stores all tokens as long as memory is available and throws away tokens if it runs out of memory. The design goal of the caching strategy is to distinguish between more and less valuable tokens. If memory runs short, the less valuable ones are the first to be discarded.

The value of a token is determined by three factors: the effort it takes to get another token for the same group, the probability that the token is needed for following join operations and the height of the token, which determines its quality. The higher these factors are, the more valuable is a token and hence should be kept as long as possible.

We divide token packets into three categories. A token packet is a first class token packet, i.e. most valuable, if it is the only token packet for the corresponding group at this repServer. This refers to the first value factor; if a first class token packet is discarded, the effort to get another token is high, since a global token search must be started.

A second class token packet is a token packet of a multicast group that is highly dynamic and/or grows very fast. In particular, this can be the case if the group has been created only recently. The definition of this second class recognizes the second value factor, the probability that a token is needed for a following join operation. However, to keep tokens of recently created groups has a second reason, too. In general, the ACK tree of a recently created group has only a few hierarchy levels. If tokens are discarded in this phase, this may result in highly unbalanced ACK trees.

Third class token packets are all remaining ones not belonging to class one or two. Third class tokens should be removed at first, then second class tokens and first class tokens last. If a class contains several token packets, always the token packet with the greatest height should be removed.

With the described caching mechanism we ensure, that if memory runs short, we keep valuable tokens as long as possible. This reduces the message overhead of the TRS and leads to well-shaped ACK trees with low height and low delays between successor and predecessor nodes. As we will be seen in the next section, our protocol is still operational even if all tokens are discarded. Hence, limited memory does not affect the robustness of the proposed protocol.

H. The Protocol in the Presence of Failures

RepServers may become unavailable due to crashes or network partitioning. Since we assume, that token basket as well as group records are stored in volatile memory due to performance reasons, these information is lost in case of crashes and will not be recovered.

When the home repServer is not available when *repCreateGroup*, *repDeleteGroup* or *repJoinGroup* is to be performed, some other leaf repServer can be selected, preferably in a close domain. The caller of *repLeaveGroup* can just give up in this case as the expiration date mechanism will ensure that the corresponding token packet will be deleted.

If the home repServer is not available, alternatively, the issuer of *repCreateGroup* and *repDeleteGroup* can just give up. In the case of *repCreateGroup* no token information is established, which is treated the same way as the loss of tokens due to crashes (see below). In the case of *repDeleteGroup* the expiration date mechanism will eventually removes all outdated tokens. To perform joining a group, the joining node can alternatively initiate ERS if its home repServer is down (see below).

If a non-leaf repServer crashes, it looses all group records. Although possible, we believe that recovering the group records after restarting is not worth the effort. Due to an unavailable or recently restarted repServer, a group's tree may be in an inconsistent state. However, the expiration date and update mechanism take care of that and after some time, each group tree is recovered. If a join group operation fails due to an unavailable repServer or inconsistent group tree, ERS must be used instead.

When ERS delivers a predecessor node, the joining node as the owner, creates a new token packet by issuing a *repAddToken*, preferably at its home repServer. With this mechanism we ensure that the next join operation concerning this group can be processed by the TRS again. After a node has joined via ERS, the predecessor

node (discovered by ERS) calls a *repRemoveToken* operation at its home repServer to remove one of its tokens in the repository.

In summary, the TRS is as robust as ERS, since it is still operational even if all repServers are unavailable.

IV. Simulations

In this section, we will present simulation results that compare TRS with expanding ring search strategies in terms of message overhead, round trip delay and reliability of the generated ACK trees.

A. Simulation Scenario

Our simulations were performed using the NS2 [15] network simulator. The networks were generated with Tiers [16] and consists of up to almost 2000 nodes. The links' bandwidth is 10Mbps for LAN links, 100Mbps for MAN links and 1000Mbps for WAN links; the link delays are chosen randomly for each link from 1ms to 3ms for LANs, 1ms to 8ms for MANs and 5 to 19ms for WANs.

Since the multicast routing protocol significantly affects the measured results, we have run most simulations with both, the distance vector multicast routing protocol (DVMRP) [12] and protocol independent multicast - sparse mode (PIM-SM) [13].

Some simulations are performed with various background traffic conditions. The background traffic is generated by randomly placed senders and receivers of TCP streams. The traffic generated by a sender is distributed exponentially. Since the background traffic consumes a lot of CPU and memory resources we were not able to simulate high background load with the given network bandwidths. Therefore, we had to decrease the bandwidth by factor 100 to be able to run the simulations with background traffic.

The TRS is configured with 8 leaf repServers and a branching factor of 2, which results in 15 repServers in total.

B. Simulation Results

Figure 6 shows the average received number of messages per repServer. Since the minimal-height strategy results in more global token searchs than the random-choice strategy, more messages must be sent within the repository tree. However, if a maximum height deviation of 2 is allowed, the message overhead already decreases significantly. For example, to process 200 join operations with deviation 0, Figure 6 depicts 59 messages for the root repServer to be received. If a deviation of 2 is allowed, the root repServer must process only 6 messages. If the repServers should be overloaded, the number of repServers must be increased to distribute the load.

Figure 7 shows the dependency between the message overhead and various levels of background load. The background load is defined to be the percentage of busy links during simulation time, i.e. if the background load is 100% each network link is busy during the entire simulation.

The timeout parameter for ERS specifies the time per hop a node waits for an answer to arrive, before it sends a new search message with an increased TTL. For example, if the timeout is 1 second and the search scope 10 hops, then the node issuing ERS waits 10 seconds for an answer before it starts a new search.

The results show that ERS scales poorly with the background load. If the background load exceeds a certain level, the number of received messages rises exponentially. This behavior is caused by increased message delays due to high background load. When the delay of a search and the resulting answer message exceeds the timeout interval, the issuer of ERS sends a new multicast message with increased TTL. The smaller the timeout interval, the earlier occurs this effect. However, the timeout parameter can only be increased within a certain range, since this affects the delay of a join operation. Moreover, as it can be seen in the chart, increasing the timeout interval also increases the message overhead in

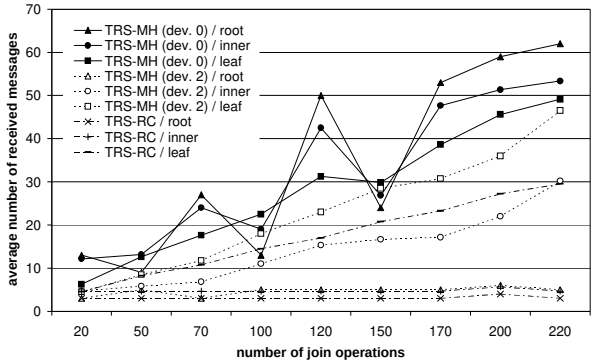

Figure 6: Number of received messages per repServer

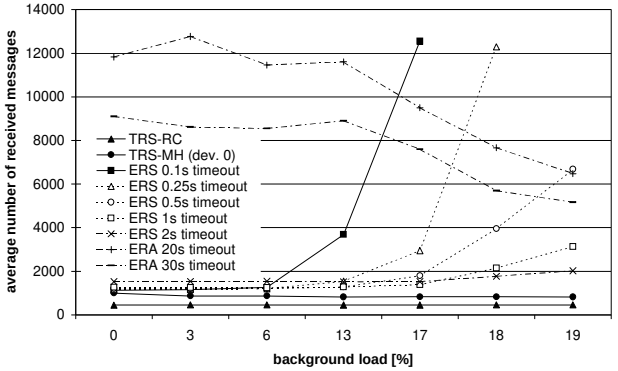

Figure 7: Scalability in terms of network load

the case of low background load. Since it takes longer for a node to join the ACK tree if the timeout interval is increased, it takes also longer before the joining node itself is able to accept successor nodes. Therefore, other joining nodes must possibly search in a larger scope to connect to the ACK tree. As it can be seen in the chart, ERA results in a high message overhead. With increased background load, the message overhead seems to decrease but this is only caused by our simulation scenario. The use of ERA leads to a high network congestion and so not all invitation messages were delivered within simulation time.

Figure 8 shows the message overhead of 50 join operations for various network sizes. TRS causes the lowest message overhead and moreover, in difference to ERS, this overhead is also independent of the network size. Note that multicast messages sent by ERS and ERA are even counted as a single message, so the real network load with these multicast based approaches is even higher as the chart indicates. As it can be further seen in the figure, TRS-MH causes more messages than TRS-RC. If we enforce strict height-balanced ACK trees, the minimal-height strategy needs about twice as much messages as the random-choice strategy. However, if we allow a maximum height deviation in the created ACK tree of two levels, the minimal-height strategy results in a only slightly increased message overhead compared to the random-choice strategy. The results show that the TRS scheme scales significantly better than ERS and ERA.

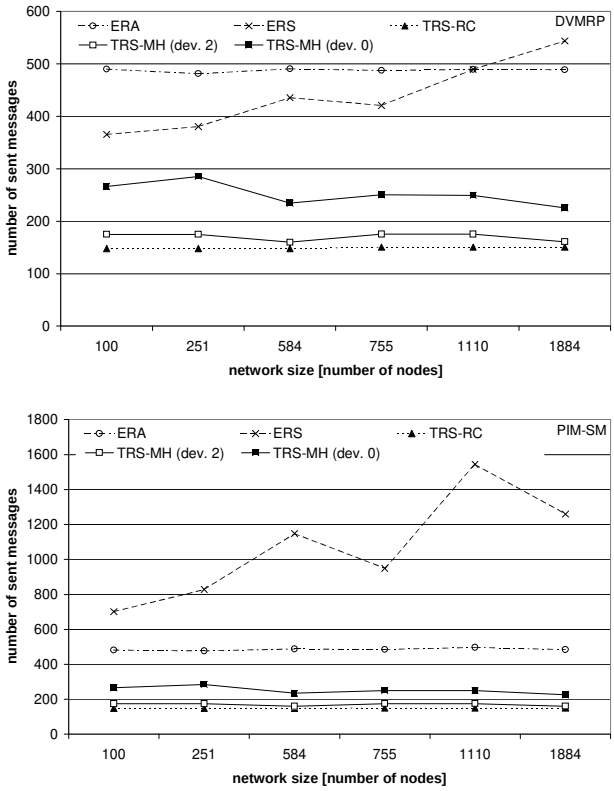

Figure 8: Scalability in terms of network size

Figure 9 shows the round trip delay depending on the number of join operations. The network consists of 251 nodes. Each dot in the chart is the average round trip delay of 12 measurements with different randomly distributed join operations and different background load levels. The round trip delay is assumed to be the time between sending a multicast message and receiving the last aggregated ACK at the root node.

The results show that TRS-MH decreases the round trip delays. The poor results of ERS and ERA with PIM-SM is conspicuous. If PIM-SM routing is used, the dissemination of multicast messages starts always at the same core node for all senders. Therefore, ERS and ERA typically finds always nodes close to this core rather than close to the searching node, which results in high round trip times.

In our last simulation, we have investigated the reliability of the created ACK tree, which is determined by its shape. As mentioned in Section II, the availability of a node depends on the availability of its ancestors in the ACK tree. If an non-leaf node becomes unavailable, this may cause message loss and extra overhead for rejoining the successors. In this respect, a well-shaped tree has a minimal number of inner nodes which is satisfied by height-balanced trees. Figure 10 shows the average num-

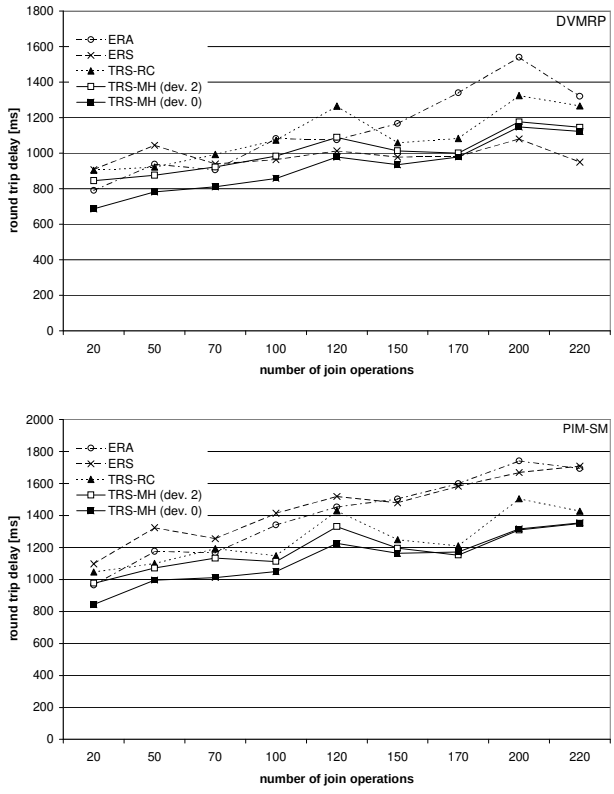

Figure 9: Round trip delay

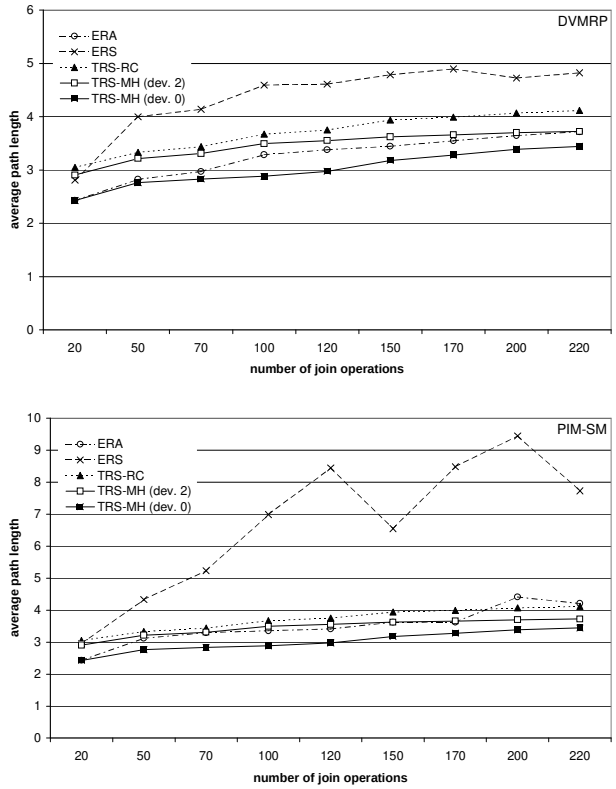

Figure 10: Average path length

ber of nodes that must rejoin the tree if a single ACK tree node fails. Note that this number is equivalent to the average path length in the ACK tree. Of course, the minimal-height strategy leads to the best results which corresponds with the theoretically achievable minimum. The use of ERS leads to a high number of dependent nodes, especially in combination with PIM-SM routing.

The presented simulations with a wide variety of networks sizes, number of receivers and background traffic illustrates that TRS-MH performs better than ERS or ERA approaches and, in terms of round trip delay and average path length, better than TRS-RC.

V. Summary

The token repository service with minimal-height strategy is a novel approach that allows to create height-balanced ACK trees. The basic concept is a distributed token repository storing tokens which represents the right to connect to a certain predecessor node in the corresponding ACK tree. We have described how it can be implemented in fault tolerant, yet efficient manner.

Simulation studies in this paper illustrates that the token repository service with minimal height strategy performs better than ERS and ERA. It results in a lower message overhead and ACK trees with higher quality in terms of round trip delay and average path length.

References

[1] S. Pingali, D. Towsley, F. Kurose: A comparison of sender-initiated and receiver-initiated reliable multicast protocols, Proceedings of ACM SIGMETRICS, 1994, pages 221-230.

[2] B.N. Levine, J.J. Garcia-Luna-Aceves: A comparison of known classes of reliable multicast protocols, Proceedings of the IEEE International Conference on Network Protocols, 1996, pages 112-121.

[3] R. Yavatkar, J. Griffioen, M. Sudan: A reliable dissemination protocol for interactive collaborative applications, Proceedings of the third ACM International Conference on Multimedia, 1995, pages 333-344.

[4] J.C. Lin, S. Paul: RMTP: A reliable multicast transport protocol, Proceedings of the Conference on Computer Communications (IEEE Infocom), 1996, pages 1414-1424.

[5] D. M. Chiu, S. Hurst, J. Kadansky, J. Wesley: TRAM: A tree-based reliable multicast protocol, Sun Microsystems Laboratories Technical Report Series, TR-98-66, 1998.

[6] M. Hofmann: Adding scalability to transport level multicast, Lecture Notes in Computer Science, No. 1185, 1996, pages 41-55.

[7] S. Deering, D. Cheriton: Host Groups: A multicast extension to the internet protocol, RFC 966, 1985.

[8] S. Deering: Host extensions for IP multicasting, RFC 1112, 1989.

[9] D. Boggs: Internet broadcasting, XEROX Palo Alto Research Center, Technical Report CSL-83-3, 1983.

[10] K. Rothermel, C. Maihöfer: A robust and efficient mechanism for constructing multicast acknowledgment trees, Proceedings of the Eight International Conference on Computer Communications and Networks (IEEE ICCCN), 1999, pages 139 - 145.

[11] A. S. Tanenbaum: Computer networks, 3rd edition, Prentice-Hall, 1996.

[12] D. Waitzman, C. Partridge, S. Deering: Distance vector multicast routing protocol, RFC 1075, 1988.

[13] Estrin, D.; Farinacci, D.; Helmy, A.; Thaler, D.; Deering, S.; Handley, M.; Jacobson, V.; Liu, C.; Sharma, P.; Wei, L.: Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification, RFC 2362, 1998.

[14] D. Clark: The design philosophy of the DARPA internet protocols, Proceedings of ACM SIGCOMM, 1988, pages 106-114.

[15] UCB/LBNL/VINT Network Simulator - ns (version 2), http://www-mash.cs.berkeley.edu/ns/ns.html.

[16] Tiers Topology Generator, http://www.geocities.com/ResearchTriangle/3867/sourcecode.html.