Universität Stuttgart
Fakultät Informatik

A Robust and Efficient Mechanism for Constructing Multicast Acknowledgment Trees

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

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

A Robust and Efficient Mechanism for Constructing Multicast Acknowledgment Trees

K. Rothermel, C. Maihöfer

Bericht 1999/07
Juni 1999

Abstract

A great variety of todays networked applications require a reliable multicast service. A number of the proposed reliable multicast protocols use a positive acknowledgment scheme, which returns ACKs to the sender to confirm correct delivery. To avoid the well-known implosion problem in the case of large receiver groups, often a tree-based approach is used, i.e., receivers are organized in a tree and ACK messages are passed along the edges of this so-called ACK tree. For building up this tree variations of the Expanding Ring Search (ERS) scheme have been proposed. However, our simulations show that ERS scales poorly.

In this paper, we propose an alternative scheme for building up ACK trees. This scheme is based on a so-called token repository service, where a token represents the right to connect to a certain node in the corresponding ACK tree. Nodes that want to join a group just request a token for this group from the (distributed) token repository service. Our simulations show that our scheme causes a much lower message overhead than ERS. Moreover, the quality of the resulting ACK trees in terms of delay and reliability is in many cases higher if generated with our scheme.

1 Introduction

For many of todays networked applications efficient support for one-to-many or many-to-many communication relationships is a prerequisite for scalability. Examples for those applications are publish&subscribe services, large scale groupware systems, video/audio broadcasting, and teleteaching. For the sake of scalability and the efficient use of network resources those applications are to be based on a multicast communication service. The IP multicast protocol [1, 2] used in the Internet provides a best effort service. However, a wide range of the existing applications require reliable multicast.

A number of reliable multicast protocols have been proposed in the literature [3, 4, 5, 6, 7, 8]. Many of them are based on IP multicast. In order to achieve reliability, a number of these proposed protocols are based on a positive acknowledgment scheme, where receivers return ACKs to the sender to confirm correct delivery. If ACK messages are sent directly to the sender, this may cause the well-known implosion problem for large receiver groups. The most promising approach to avoid this problem, is to organize the group of receivers in a tree [5, 6, 7, 8]. While multicast messages are sent directly (e.g., via IP multicast), ACK messages are collected and propagated along the edges of the so-called ACK tree.

When a new node wants to join a multicast group it has to be connected to the group's ACK tree. For finding an appropriate node in the existing ACK tree it has been proposed to use a technique called Expanding Ring Search (ERS) [9]. The advantage of ERS, which is based on IP multicast, is its robustness. On the other hand, its main disadvantage is the huge amount of messages generated as well as the problems associated with the various IP multicast routing protocols.

In this paper we propose an alternative approach for constructing ACK trees. The proposed approach is based on the concept of a (distributed) token repository. Basically, a token represents the right to connect to a certain node in an existing ACK tree. When a node wants to connect to a group's ACK tree, it asks the token repository for a token of that group. The token delivered to this node identifies its predecessor in the ACK tree. In order to allow other nodes to connect to the new member later, *k* tokens are generated for this member in the token repository, where *k* is the number of successors the node can accept in the ACK tree.

Our simulation results show that the proposed token repository scheme scales much better than ERS. With ERS, the message overhead may increase exponentially, whereas the message overhead of the token repository approach is constant and much lower compared to ERS. The resulting ACK tree impacts both delay of ACK messages and the reliability of multicast. In terms of delay our scheme performs almost as good as ERS based on DVMRP routing [10] and much better than ERS based on CBT [11]. In terms of reliability, our scheme performs always better.

The remainder of this paper is organized as follows. In the next section, related work is discussed. Section 3 gives an overview of our approach and Section 4 describes the implementation of the token repository service in detail. Section 5 shows simulation results before we conclude with a brief summary.

2 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. A 0-bounded node is always occupied and thus can be a leaf node only. The bound chosen for a node depends on various characteristics, such as the node's reliability, load, and performance. Therefore, we assume that different nodes may choose different bounds.

When a new member joins the group, it must be connected to a non-occupied node in the corresponding ACK tree. Generally speaking, the problem is to establish an low delay ACK tree in an efficient manner. The motivation for ACK trees causing low delays stems from the observation that average throughput of a reliable (multicast) channel based on the PAR scheme [12] is limited by the quotient of buffer size/round trip time. In other words, the lower the delay the higher the throughput for a given buffer size respectively the lower the required buffer size for a given throughput.

To construct ACK trees only variations of ERS have been proposed in the literature so far [9, 5]. ACK tree construction with ERS, which is based on IP multicast [1, 2], is rather simple. When a node wants to join a group, it simply multicasts a join request to the members of this group. In order to decrease the network load and to find a "close" predecessor the multicast is limited in scope. The time-to-live field (TTL) in the IP header is set to one for the first join request, i.e., the scope of the multicast is limited to the LAN of the sender. When a node receives this request and it is not yet occupied it returns an answer. If no node answers within a certain time, the node that wants to join the group multicasts a new join request with an increased TTL of two. This is repeated until an answer message arrives or the maximum TTL of 255 is reached. The node that answers first becomes the predecessor of the new member.

There is one major variation to the described method. In [13] not only the joining receivers use ERS to find a predecessor node but also the nodes that are already in the ACK tree use ERS to send invitation messages to new nodes that want to join the tree. The advantage of this approach is that the underlying network need not support bidirectional multicast service, the disadvantage is the overhead due to the invitation messages, which are sent even if no node wants to join.

The great advantage of ERS is fault tolerance. The main disadvantage of ERS is the large amount of messages particularly in sparse, widely distributed groups. Another drawback of ERS is its dependency on the various IP multicast routing protocols, each of which has its own problems. ERS with DVMRP routing leads to a vast overhead at all involved routers because a new multicast tree is to be build for each sender, i.e, a individual multicast tree has to be built up for each node joining the group. Note that these trees are only used for running ERS. With shared tree approaches like CBT or PIM-SM, the use of ERS results in a traffic concentration at the core, still a vast amount of ERS messages, and higher ACK delays.

3 The Token Repository Service

In this section, we will sketch the concept and interface of a token repository service, which is suggested to be the basic infrastructure for building up ACK trees. The implementation of this service will be covered in the next section.

Remember, a token represents the right to connect to a particular node in a given ACK tree. When a k-bounded node joins a group, *k* tokens are generated and stored in the token repository, where the joining node is called the ***tokens' owner***. A token is identified by a pair <*Group*, *Owner*>, where *Group* is the group the *Owner* of the token belongs to. If *Owner* is k-bounded in *Group*'s ACK tree, then initially there exist *k* tokens with identifier (*Group*;*Owner*) in the repository. 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 token repository service.

- repCreateGroup (Group, ACK_Root, K):
  This operation makes the group identified by *Group* known to the repository service. The root of *Group*'s ACK-tree is identified by *ACK_Root*, which is *K*-bound.
- repDeleteGroup (Group):
  This operation deletes all state information associated with *Group* in the repository.
- repJoinGroup(Group,NewMember, K) returns (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 token repository.
- 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 repository.
- repRemoveToken(Group,Owner):
  This operation is called when node identified by *Owner* in *Group*'s ACK tree accepts a new successor via a ERS join request. It removes one of *Owner*'s tokens.

TABLE 1. Operations provided by the token repository service

For the token repository service, we had the following design goals in mind:

Well-shaped ACK trees: The shape of the ACK tree impacts both the delay of ACKs as well as reliability. In the case of delay not only the depth of the tree but also the delay associated with each particular edge has to be considered. 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 used as a quality criteria for reliability.

Efficiency: The service should generate only little communication and processing overhead. In particular, token information and all other group state information is to be stored in volatile memory only. The overall goal is to cause a significantly lower message overhead than ERS.

Scalability: Even for a huge amount of groups and large groups the token repository service should not become a bottleneck. To achieve this we will use a hierarchy of token repository servers.

Robustness: The token repository service should be available even in the presence of node crashes and network partitions. The goal is to achieve the same degree of robustness as ERS.

4 Implementation of the Token Repository Service

In this section, we will first present the basic principles of the implementation. Then we will describe, which state information is maintained by the token repository. Subsequently, we will describe the protocol operation for creating, deleting, joining and leaving groups during normal operation, i.e., when describing the protocols we assume the absence of failures. Finally, we will consider communication and node failures in the last part of this section.

4.1 Basic Principle

A prerequisite of our approach is that the network is structured into ***hierarchical domains***. The root domain includes all nodes of the network, while the leaf domains encompasses disjunct set of nodes. Inner domains contain the nodes of their successor domains. It is our assumption that domains group the nodes of the network by communication distance. In other words, the distance between two nodes belonging to the same domain is typically smaller than the distance between two nodes in different sibling domains. Figure 1 shows the association of nodes to domains.

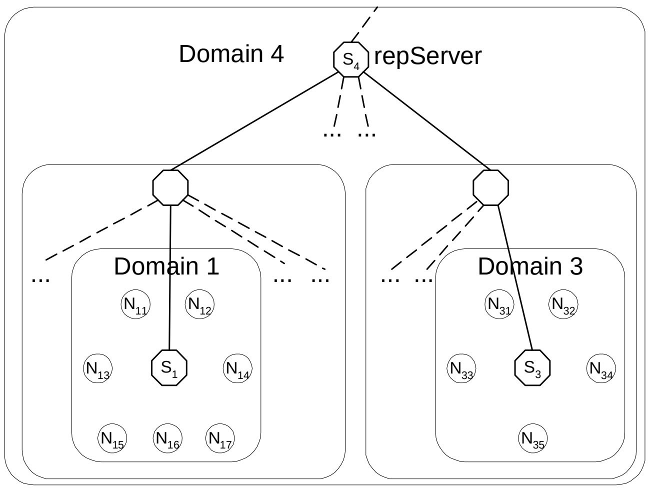

FIGURE 1. Domain structure

The repository service is implemented as a distributed system, consisting of a hierarchically structured set of ***token repository servers*** (***repServers*** for short), where there exists one repServer for each domain in the domain hierarchy. That is, the predecessor of a repServer responsible for a domain, say *D* is the repServer being in charge for *D*'s predecessor in the domain hierarchy. For example repServer S_1 in Figure 1 is responsible for domain 1 and repServer S_4 is responsible for domain 4 which consists of several subdomains.

Nodes access the token repository service only via leaf repServers. A node residing in given leaf domain contacts the repServer responsible for this domain, which we will call the node's ***home repServer***. In Figure 1 S_1 is the home repServer for all nodes N_{1x} of domain 1.

All token information is stored on leaf repServers in so-called ***token baskets***. On a leaf repServer there exists on basket for each known group, which includes all tokens of the corresponding group. Of course, several leaf repServers may store token baskets for the same group, and a leaf repServer may store token baskets of a number of groups.

To facilitate searching for token information 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 token baskets of this group and all ascendants of these nodes in the repServer hierarchy. Group tree information is stored in ***group records*** on non-leaf repServers.

Figure 2 illustrates a scenario, where a node creates a group and two other nodes join this group. Node N_{11} creates group G by sending a *repCreateGroup* request to its home repServer S_1. Upon receipt of this request, S_1 creates *k* tokens <G, N_{11}> in G's token basket, assuming that N_{11} (i.e.

the root of G’s ACK tree) will accept at most *k* successors. Moreover, group records are established in each non-leaf repServer along the path from S_1 to the root repServer.

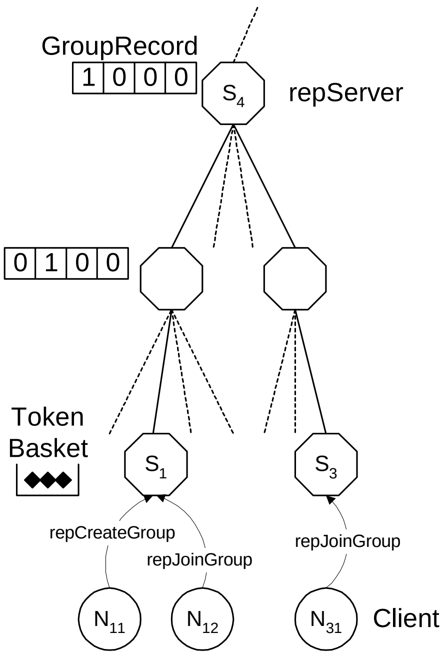

FIGURE 2. Scenario for create and join operations

N_{12} resides in the same domain as N_{11}, and hence sends its *repJoinGroup* request also to S_1. When receiving this request, S_1 checks whether or not tokens for G are locally available. It finds a token <G, N_{11}> and delivers it to N_{12}. In addition, it creates *l* tokens <G, N_{12}> in G’s token basket, assuming that N_{12} will accept at most *l* successors. After receiving token <G, N_{11}>, N_{12} can connect to N_{11} in G’s ACK tree.

When N_{31} wants to join G, it contacts its home repServer S_3, which notices that no tokens are locally available for G and hence starts a token search operation. With this search mechanism, the repServer (recursively) asks its predecessor until the first repServer, say S_4 is found that belongs to G’s group tree, i.e., stores a group record for G. G’s group tree information stored in non-leaf repServers is used to find the path from S_4 down to S_1. A token <G, N_{11}> is selected and delivered to N_{31}, group records for G are established on the path from S_4 to S_3, and finally *m* tokens <G, N_{31}> are generated and stored in G’s token basket on S_3. The next *repJoinGroup* request concerning G can be served by S_3 without involving a search operation.

In summary, if a requested token is locally available at the requester’s home repServer, the requester and the owner of this token are in the same leaf domain, i.e., this is the best case in terms of communication distance between requestor and owner. If a token is not locally available, the search procedure tries to find a token in the domain of the home repServer’s predecessor. If a token is not available in this domain either, search continues within the encompassing domain, and so on. Note that this search procedure together with the assumption that the domain hierarchy groups nodes by communication distance makes sure that a token is selected, whose owner is as close as possible to the joining node.

Due to performance reasons, token baskets as well as group records are stored in volatile memory. Consequently, repServer crashes may cause the loss of token and group information. In rare situations it may happen that the standard mechanisms for creating and joining groups temporarily fail due to crashes or network partitioning. In those situations, the repository service switches to ERS. RepServers can recover from failures without any extra effort. Since we can count on ERS as fall back mechanism, we were able to design light-weight protocols for normal operation. Moreover, the token repository mechanism provides the same level of robustness as ERS.

4.2 Group State Information

As mentioned above, leaf repServers store tokens in ***token baskets***, where there exists one token basket for each group known at the repServer. A token basket has the following structure:

- *Group*: This field uniquely identifies the corresponding multicast group.
- *SetofTokenPackets*: The tokens in the token basket are grouped by owners into so-called token packets.

Each token packet includes the following information:

- *Owner*: This field identifies the owner of the token packet uniquely. A node receiving a token out of this packet is allowed to connect to Owner in the corresponding ACK tree.
- *Height*: This field specifies the height of Owner in the corresponding ACK tree. This information can be used to distinguish the “quality” of alternative tokens. Tokens with low height are preferable if the goal is to minimize the height of the ACK tree.
- *Tokens*: This field specifies the amount of tokens in this token package.
- *ExpDate*: The field defines when the token packet expires (see below).

Group tree information is stored in ***group records*** on non-leaf repServers, where a repServer only stores a group record of a group, say *G* if one of its (leaf) descendants store tokens of *G*. A group record of repServer *S* includes the following fields:

- *Group*: Identifier of the corresponding group.
- *Succ*: This is a bitmap encoding the list of S’s successors in the group tree.
- *ExpDate*: This field defines when the group record expires (see below).

Of course, a group tree may grow and shrink during its lifetime, i.e., group records are to be created, updated and deleted dynamically. Two operations cause the group tree of a group to be modified: the creation and deletion of a token basket of this group on a leaf repServer. A token basket is to be established when the first set of tokens associated with the group are created, and it is deleted when the last token has been removed from it. When a token basket is created, the repServer performing this operation connects to the group’s group tree by sending a *TokenAvail* message to its predecessor in the repServer hierarchy. This message is forwarded in a leaf-to-root direction until it is received by a repServer that already stores a group record. All non-leaf repServers forwarding this message establish the corresponding group record. When a token basket is removed a *NoTokenAvail* message is sent to the repServer’s predecessor. A non-leaf repServer receiving this message checks whether the message’s sender was its only successor in the group tree. If this is the case, it deletes the group record and forwards the *NoTokenAvail* message to its predecessor, if any. Otherwise, it just removes the message’s sender from the group record’s *Succ* list.

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 basked is deleted and a *NoTokenAvail* message is sent to the predecessor of the repServer. When a group record expires, this repServer storing this record asks the repServers identified by the *Succ* list included in the expired record whether they are still part of the corresponding group tree. If at least one of them responds yes, then the record's expiration date is extended accordingly, otherwise it is deleted and a *NoTokenAvail* message is sent the repServer's predecessor, if any.

4.3 Creating and Deleting Groups

When a new group is created, an initial group tree must be established. Assume that S is the leaf repServer which the *repCreateGroup* operation was issued at. The group tree to be established consists of S and all ancestors of S in the repServer hierarchy. When *repCreateGroup(Group*[*], *Ack_Root, K)* (see Table 1) is issued, S creates a token basket for *Group* with the amount of tokens specified by *K*, where *Ack_Root* becomes the owner of these tokens. Subsequently, S sends a *CreateGroup* request to its predecessor. When receiving a *CreateGroup* request, a non-leaf repServer creates and initializes a group record and sends a *CreateGroup* to its predecessor, if any.

When the operation *repDeleteGroup(Group)* is issued at a leaf repServer, this server deletes the *Group*'s token basket 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 the expiration mechanism ensures that all state information is removed within a certain time despite of node and communication failures.

4.4 Joining and Leaving Groups

When a *repJoinGroup*(*Group*, *NewMember*, *K*) *returns* (*Token*) operation is called, the called leaf repServer, say *S*, checks whether tokens for *Group* are locally available. If this is the case, *S* removes a token from the token packet with the smallest *Height* value and returns this token to the caller. It also generates a new token packet for *NewMember* with *K* tokens and puts it into *Group*'s token basket.

* Here we assume an external mechanism that ensures group ids to be unique. In other words, we assume that a *repCreateGroup* operation is called only once for a given group id. If group ids are reused, we assume that by the time a group id is reused all state information concerning this id has been removed from all repServers. Note that the expiration date associated with state information makes it possible to determine how long it takes in the worst case until all state information has been removed after deletion of a group.

If no token basket exists for *Group* instead, *S* starts the token search procedure which proceeds in two phases, the upward and downward search phase. *S* initiates the upward search phase by sending a *TokenSearch* request to its predecessor in the repServer hierarchy. This request is forwarded in a leaf-to-root direction in the repServer hierarchy until a repServer is found that is part of *Group*'s group tree (i.e., stores the corresponding group record) and has at least one successor in this tree being not on the path to *S*. This repServer initiates the downward search phase by sending a *TokenSearch* to one of its successors in *Group*'s group tree. This request if forwarded along the edges of this group tree in a root-to-leaf direction until it arrives at a leaf repServer. If a repServer has more than one successor in the group tree it randomly selects one of them for forwarding the request. The leaf repServer receiving the *TokenSearch* request removes a token with the lowest *Height* value from *Group*'s token basket and delivers the token directly to *S*[*].

When receiving the token, *S* establishes a token basket, includes a token packet for *NewMember* as described above, and delivers the token to the caller of *repJoinGroup*. In order to connect itself to *Group*'s group tree, it sends a *TokenAvail* message to its predecessor (see Sec. 4.2).

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 is removed and a *NoTokenAvail* message is sent *N*'s predecessor (see Sec. 4.2).

4.5 Protocol in the Presence of Failures

RepServers may become unavailable due to crashes or network partitioning. Since all token information and group records are stored in volatile storage, group state information is lost due to crashes. We only assume the successor and predecessor information of the repServer hierarchy to be stored on stable storage, while group tree information (i.e., group records and token baskets) is lost due to crashes.

When the home leaf repServer is not available when *repCreateGroup*, *repDeleteGroup* or *repJoinGroup* is to be performed, some other leaf repServer can be selected to execute those operations. When selecting the server, clearly those that are in "close" domains are preferable. *repLeaveGroup* has to be issued at the repServer that performed the corresponding *repJoinGroup* operation. If this server is unavailable the issuer of *repLeaveGroup* can just give up as the expiration mechanism will ensure that the corresponding state information will be deleted.

Alternatively, the issuer of *repCreateGroup* and *repDeleteGroup* can just give up if its home repServer is not available. In the case of *repCreateGroup* no state information is established, which is treated in the same way as the loss of state information due to crashes (see below). In the case of *repDeleteGroup* the expiration mechanism takes care of the removal of state information. The caller of *repJoinGroup* can alternatively initiate ERS if its home repServer is down (see below).

[*] Due to failures or temporary inconsistencies it might happen that search does no succeed. For example, during the upward search phase no member of the corresponding group tree might be found or the leaf repServer receiving the search requests might have no tokens for the corresponding group. If the search procedure fails for some reason the repServer initiating the search procedure will time out and initiate ERS (for details see Sec. 4.5).

Token baskets lost due to crashes will not be recovered. When a *repJoinGroup* operation is performed after a crash, this operation will cause a search for tokens in the repServer hierarchy as described in the previous section. If no token is found for the corresponding group, ERS is applied (see below). Consequently, the effect of lost tokens is limited to nodes that are not fully occupied.

Now let us consider failures affecting the availability of non-leaf repServers. Assume that non-leaf repServer *S* becomes unavailable. In case of *repCreateGroup*, *repDeleteGroup* and *repLeaveGroup* operations, the group tree information update (if needed) is performed up to S and then terminates. This may leave the tree information temporarily in an inconsistent state, however for the latter two operations the expiration mechanisms takes care of that. In the case of *repCreateGroup*, the termination of the update process is analogous to a crash of S and its ancestors (see below). A *repJoinGroup* operation succeeds if a token can be found before the search procedure hits *S*. Otherwise, search gives up and ERS is initiated (see below).

If a non-leaf repServer crashes, it looses all group records, i.e., S's subhierarchy becomes disconnected from the corresponding group trees. Although group records could be recovered just by contacting the successors after restart we decided not to do so since we think it is not worth the effort. As a result of a lost group record, it may happen that the most appropriate token is not found by the search procedure. In the worst case, no token is found even if tokens are available in the disconnected part of the tree and ERS must be used instead. Note that the tree update process described in Sec. 4.2 establishes new group records when new token baskets are created in the disconnected hierarchy.

Finally, let us consider the effects of ERS, which is initiated by the corresponding leaf repServer whenever the search procedure is unable to find a token. When ERS delivers an identifier of a node, say *N*, a token identifying *N* is returned to the caller of *repJoinGroup*(*Group*, *NewMember*, *K*) *returns* (*Token*). Moreover, a token basket including a token packet for *NewMember* is created, and a *TokenAvail* message is sent to the repServer's predecessor (see Sec. 4.2). Note that the next join operation concerning *Group* can be processed locally.

In summary, the proposed mechanism is still operational even if all repServers are unavailable since ERS is used as a fallback mechanism. Consequently, it is as robust as ERS.

5 Simulations

We have simulated ERS and the token repository scheme using the NS2 network simulator. Our simulation scenarios consist of hierarchical networks, including LAN, MAN and WAN structures.

Figure 3 shows the round trip delay depending on the number of join operations for the token repository service and ERS with DVMRP and CBT routing. In our simulations, between 20 to 220 nodes join the ACK tree in a network consisting of 251 nodes in total. 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. Each dot in the figure is the average of 12 measurements with different randomly distributed join operations and different background loads. The results show that the token repository service only leads to slightly higher round trip times compared to ERS with DVMRP. If CBT routing is used, the dissemination of multicast messages starts always at the same core node for all senders. Consequently, ERS typically finds nodes close to the core rather than the searching node. This fact results in poor round trip times as depicted in Figure 3. Note that the token repository scheme leads to a significantly lower round trip delay as ERS with CBT.

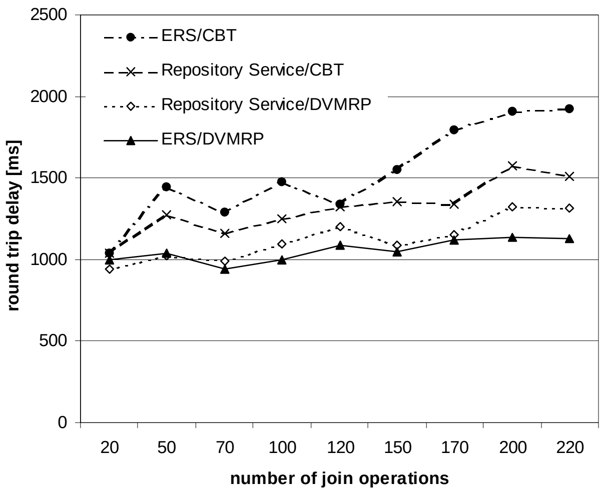

FIGURE 3. ACK tree round trip delays

Figure 4 and Figure 5 show the results of our investigations concerning scalability of ERS and the token repository scheme. Figure 4 shows the message overhead for 50 join operations dependent on different network sizes. Our simulations show that the token repository service not only causes the lowest message overhead but this overhead is also independent of the network size. It is important to say that each multicast message sent by ERS has been measured as a single message. Consequently, we compare unicast messages - in the case of the token repository scheme - with multicast messages - in the case of ERS. While ERS with DVMRP routing causes a doubled message overhead compared to our scheme, ERS with CBT routing causes at least four times more messages. Again the reason for this effect is that multicast dissemination always starts from the same core node. The ACK tree nodes close to the core are soon occupied, and hence large search scopes are needed to discover appropriate nodes.

Figure 5 shows the dependency between (received) messages and various levels of background load for our scheme and ERS with DVMRP. The timeout parameter for ERS specifies the time a node waits for an answer to arrive before it sends a new search message with an increased TTL. Note that the timeout parameter specifies the waiting time per hop, e.g., if the timeout is 1 second and the search scope is 10 hops then the node issuing ERS waits 10 seconds before it starts a new search. If ERS is used and the background load is high, the number of received messages rises exponentially at a certain percentage of background load, and the smaller the timeout interval the earlier this effect occurs. Increasing the timeout parameter over a certain value is not practical anymore as this parameter impacts the delay of join operations. Moreover, as can be seen in the chart, increasing the timeout interval also increases the message overhead in the case of low background traffic. Since it takes longer for a node to join the ACK tree, 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. We can conclude from these results that the token repository scheme scales significantly better than ERS.

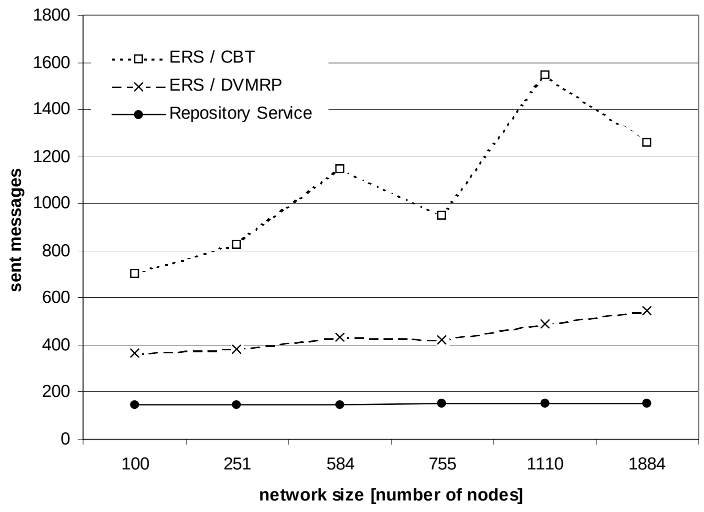

FIGURE 4. Messages sent depending on the network size

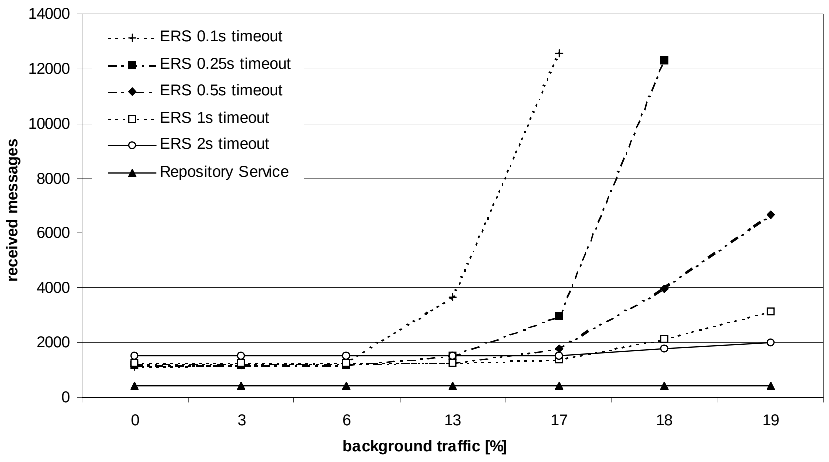

FIGURE 5. Messages received depending on the background traffic

As mentioned above the shape of the ACK tree also impacts reliability. Remember that a node depends on the availability of its ancestors in the ACK tree. If an inner node becomes unavailable, this may cause message loss and extra overhead for rejoining the successors. The optimum in this respect is a tree with the same height as a balanced k-bounded tree. Those trees are desirable, since the number of inner nodes are minimal. Figure 6 shows the average number of nodes that must rejoin the tree if a single ACK tree node fails. This number is equivalent to the average path length in the ACK tree. Our simulations show, that our token repository scheme produces ACK trees that come close to balanced trees. The use of ERS leads to ACK trees with an higher average path length.

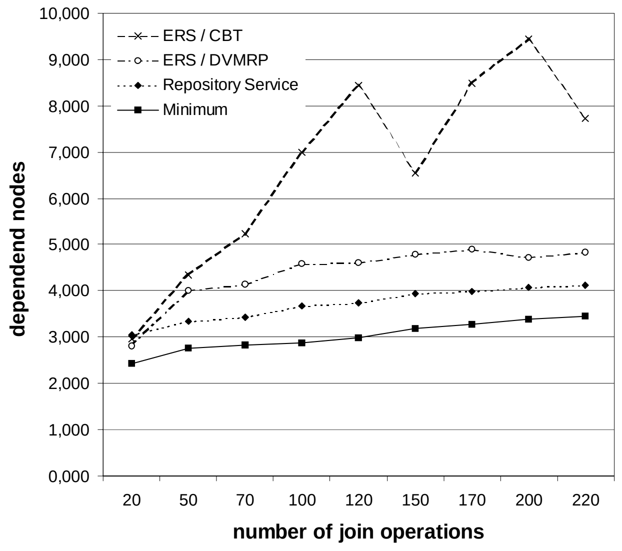

FIGURE 6. Average ACK tree path length

The presented results illustrate the great dependence of ERS on the underlying multicast routing protocol, which leads to great variations in delays, scalability and reliability. They also show that the proposed token repository scheme performs better than ERS, in particular in terms of message overhead, scalability and reliability.

6 Summary

In this paper, we have introduced the token repository service, which is a novel approach to construct ACK trees needed for reliable multicast protocols. The basic idea is to use a distributed repository. The repository stores tokens, which represent the right to connect to a certain node in an existing ACK tree. We have described how such a repository can be implemented in a fault tolerant and efficient manner. In simulation studies, the performance of the token repository service was compared to the well-known ERS approach.

Our results show, that ERS has a limited scalability and hence should not be used for constructing large multicast groups. In contrast to ERS, the token repository service has a low message overhead and is predestined even for large groups and networks. Further simulations have shown that the token repository service produces well-shaped ACK trees with low delays and short path lengths.

7 References

[1] Deering, S.; Cheriton, D.: Host Groups: A Multicast Extension to the Internet Protocol, RFC 966, 1985

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

[3] Pingali, S.; Towsley, D.; Kurose, F.: A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols, Proceedings of ACM SIGMETRICS, 1994, pages 221-230

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

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

[6] Lin, J.C.; Paul, S.: RMTP: A Reliable Multicast Transport Protocol, Proceedings of the Conference on Computer Communications (IEEE Infocom), 1996, pages 1414-1424

[7] Chiu, D. M.; Hurst, S.; Kadansky, J.; Wesley, J.: TRAM: A Tree-based Reliable Multicast Protocol, Sun Microsystems Laboratories Technical Report Series, TR-98-66, 1998

[8] Hofmann, M.: Adding Scalability to Transport Level Multicast, Lecture Notes in Computer Science, No. 1185, 1996, pages 41-55

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

[10] Waitzman, D., Partridge, C., Deering, S.E.: Distance Vector Multicast Routing Protocol, RFC 1075, 1988

[11] Ballardie, A.: Core Based Trees (CBT version 2) Multicast Routing, RFC 2189, 1997

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

[13] Levine, B.N.; Lavo, D.B.; Garcia-Luna-Aceves, J.J.: The case for reliable concurrent multicasting using shared ACK trees, Proceedings of the fourth ACM International Conference on Multimedia, 1996, pages 365-376

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