Scalable and Reliable Multicast ACK Tree Construction with the Token Repository Service

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

Abstract

Reliable multicast is realized in a scalable way by tree based approaches, where the receivers are organized in an ACK tree. Usually, expanding ring search (ERS) is used to create such ACK trees. However, ERS has some shortcomings like poor scalability, strong dependency on the multicast routing protocol and the need for bidirectional multicast capable networks, which makes it difficult to use ERS as the Internet standard mechanism.
In this paper we propose the Token Repository Service (TRS), which is based on a token repository and a modification of ERS. The TRS stores tokens, which represents the right for a joining node to connect to a certain parent node in the ACK tree. Performance evaluations show that the TRS approach has several advantageous compared to ERS, like improved scalability and independence of the routing protocol.

1. Introduction

Several protocols have been proposed to provide a reliable service on top of IP multicast [1, 2, 3, 4, 5, 6]. Reliable multicast protocols are based on the concept of controlling the successful delivery by some kind of acknowledgments returned by the receivers to the source. While simple approaches like the class of sender- and receiver-based reliable multicast protocols [1, 2] can cause the well-known acknowledgment implosion problem, tree-based approaches are the most promising ones [3, 4, 5, 6]. All group members are organized in a tree, the so-called ACK tree. Instead of sending positive or negative acknowledgments directly to the sender, each receiver confirms the correct message delivery only to its parent in the ACK tree, which is in most cases responsible for possible retransmits. Since the maximum number of child nodes is limited for each node, no ACK tree node is overwhelmed by acknowledgment messages.

However, tree-based protocols raise another problem, namely that of setting up and maintaining the ACK tree. A new member joining a multicast group must be connected to the group's ACK tree, which is usually done by a technique called expanding ring search (ERS) [7]. ERS is a multicast-based search technique for discovering a suitable parent node in the ACK tree, by gradually increasing the search scope. The advantage of ERS is its simplicity and robustness against node and network failures. On the other hand, the suitability of ERS on a large scale has not been investigated so far. Our simulation results will show that the use of ERS has several shortcomings, since it results in a large message overhead and has further special problems with the common multicast routing protocols. ERS with the source-based DVMRP [8] results in a large message overhead on the routing layer, since for each receiver joining the ACK tree a new routing tree has to be established. ERS with the unidirectional core-based PIM-SM protocol [9] results in ACK trees of poor quality, since the search for a parent node is always started from the same core node.

In this paper we present the token repository service (TRS) as an alternative approach for constructing ACK trees. Our approach is based on a distributed token repository and the traditional ERS. This provides improved scalability in the ACK tree construction and better shaped ACK trees, necessary for ensuring reliable multicasting with high throughput. The TRS stores tokens, where a token provides basically the right to connect to a certain parent node in the ACK tree. A node joining a group asks the TRS for a token of this group, which identifies the parent to connect to.

The remainder of this paper is organized as follows. In the next section background and related work are discussed. Section 3 describes the token repository service in detail and the following section describes the protocol in the presence of failures. In Section 5 we present performance evaluations before we conclude with a brief summary in Section 6.

2. Problem definition and related work

The following tree terminology is used to describe the ACK tree. If the last edge in a tree on the path from the root r to a node x is (y, x) then y is the *parent* of x and x is a *child* of y. A node with no children is a *leaf* node. All other nodes are called non-leaf nodes. The root node has *height* 1 in the tree. The *height* of any other node in the tree is one higher than the height of its parent. The *height of a tree* is the maximum height of the tree's nodes. The maximum number of children of a node x in a tree is called the *bound* of x. If a node x is k-bounded and has already k child nodes, x is called *occupied*. A *k-ary* tree is a tree with bound k for all nodes.

When a new member joins a reliable multicast group it must be connected to the group's ACK tree. In more detail it must be connected to a k-bounded parent that is not already *occupied*. Most approaches to establish an ACK tree are based on expanding ring search (ERS) [7]. With the basic ERS approach for setting up ACK trees, the joining node looks for a parent in the ACK tree by sending multicast search messages with increasing search scopes [3]. The first message is sent with a time-to-live (TTL) of one, i.e. it is limited to the sender's LAN. If a non-occupied group member receives this message it returns an answer allowing the new member to connect to it. If no node answers within a certain time, the TTL is increased and a new search message is sent. The joining node repeats this until an answer arrives or the maximum TTL of 255 is reached. Note that increasing the TTL step by step reduces the network load and detects preferably parents that are close to the searching node.

Some protocols reverse the method described above by making the non-occupied ACK tree nodes search for child nodes with multicast invitation messages [4, 5], which we will call expanding ring advertisement (ERA). The invitation messages can be sent with fixed TTL [4], increasing TTL [6] or according to a special distribution function [5]. Some protocols use a combination of both approaches [6].

ERS and ERA approaches have the great advantage of fault tolerance. On the other hand, our performance evaluations in Section 5 will show that both result in serious drawbacks, e.g. huge message overhead. An additional drawback is their dependency on the various routing protocols, resulting in particular problems with each of them. Please see Section 5 for details.

In the following sections we will describe the token repository service with proxy server strategy (TRS-PS) [10], which is based on a combination of a repository service and the well-known ERS. In [11] we have already proposed an alternative approach for creating multicast ACK trees, which we will further reference as token repository service with random-choice strategy (TRS-RC). Both strategies share the same concept which we will describe in Section 3.1 but differ in their implementation strategy. In contrast to TRS-RC, the proxy server strategy presented in this paper, uses ERS which results in less state information to be maintained and in a simpler implementation.

3. The Token Repository Service with Proxy Server Strategy (TRS-PS)

3.1 Interface and concept of the Token Repository Service

Before we describe our protocol in more detail we introduce the concept of the token repository service. A token stored in the TRS represents the right to connect to a particular parent node in a given ACK tree. When a k-bounded node has created or joined a group, *k* tokens are generated and stored in the repository. The creating or joining node is called the tokens' *owner*. A token is defined by a 3-tuple <group, owner, height>, where group identifies the multicast group of the owner. We define the height of a token to be the height of its owner in the corresponding ACK tree. The height is used to determine the "quality" of a token (see Section 3.3).

Initially, there are *k* tokens of a group in the repository, generated on behalf of a create group operation. When a node *N* wants to join a given group, it asks the TRS for a token of this group. The repository service then selects a token of this group and returns it to *N*. The returned token is removed out of the repository and new tokens with owner *N* are created. The joining node *N* is now able to connect to the received token's owner in the corresponding ACK tree.

When a node leaves a group, it removes all of this group's tokens out of the repository for which it is the owner. The leaving node has allocated a token belonging to its parent in the ACK tree. This token is returned to the repository, which then can be reused by some other node joining this group later. Table 1 shows the operations provided by the token repository service.

3.2 Implementation of the Token Repository Service with Proxy Server Strategy

In this section, we will describe the basic principles of implementing TRS-PS. To meet the design goals of scalability and reliability, the token repository service is implemented as a distributed system of token repository servers, repServers for short. Each repServer is responsible for a disjunct set of nodes, called domain. For example repServer S_1 in Figure 1 is responsible for domain 1 consisting of nodes N_{1x}.

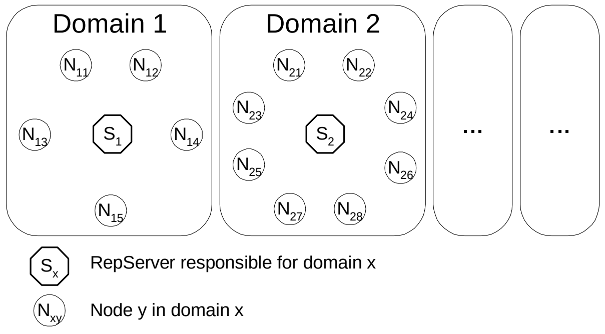


Fig. 1. Domain structure

Note that nodes can be arbitrarily assigned to domains and that the repServer does not have to be located inside its domain. However, in order to construct low-delay ACK trees and to minimize communication overhead, domains should structure the network by communication distance, i.e. the communication distance between two nodes in the same domain should be typically smaller than between two nodes in different domains. Furthermore, the repServer should be located inside its domain. A repServer, responsible for all nodes in its domain, is called these nodes' home repServer. In Figure 1, S_1 is the home repServer of nodes N_{1x}. During normal operation nodes access the token repository service only via their home repServer.

Tokens are stored at the repServers. All token information is stored in so-called token baskets. A repServer holds one basket for each known group, which contains this group's tokens. However, not all tokens of a group are stored at only one repServer, thus several repServers may store token baskets for the same group.

Tokens are created on behalf of a create or join group operation, since the creating or joining node is able to accept child

Table 1. Operations provided by the TRS

Operation	Description
repCreateGroup (Group, K)	This operation makes *Group* known to the repository service. The caller becomes the root of the ACK tree, which is *K*-bounded.
repDeleteGroup (Group)	This operation deletes all token information of *Group* in the repository.
repJoinGroup (Group, NewMember, K) returns (Token)	*repJoinGroup* is called when the node identified by *NewMember* wants to join *Group*, where *NewMember* is *K*-bound. The operation returns a token identifying the parent in the ACK tree to connect to.
repLeaveGroup (Group, Member)	This operation deletes all of *Group*'s tokens owned by *Member*.
repAddToken (Group, Owner)	*repAddToken* adds a new token owned by *Owner*. It is called by *Owner* when a child of *Owner* disconnects from the *Group*'s ACK tree.
repRefreshToken (Group, Owner, Amount)	To provide fault tolerance, *repRefreshToken* is periodically called by the tokens' owner. It indicates how many child nodes (*Amount*) can still be accepted (see Section 4).

nodes. A token is always stored at the repServer responsible for its owner's domain. If a node requests a token from its home repServer and this repServer possesses a token of the requested group, it simply delivers such a token. A repServer possesses tokens of a group only if a node in its domain has created or joined this group before. Consequently, it is possible that a repServer does not possess tokens for each group. Therefore, it is possible that a node's home repServer cannot satisfy a token request although another repServer could provide a suitable token. For example assume that all tokens of a group are stored on a single repServer S_1, responsible for domain 1. If a node in another domain, say domain 2 for example, requests its home repServer S_2 for a token, our approach must ensure that finally S_2 can deliver one of S_1's tokens to the requesting node.

To meet this requirement in the proxy server approach, a repServer initiates a token search for a group's token if a node in its domain requests a token and none is available locally. The token search is processed by ERS. All repServers belong to the same well-known multicast group. If a repServer has to search for a token, it starts an ERS search on the repServers' multicast group. If a repServer receives such a token search message and possesses a token of this group, it hands over one token to the searching repServer. In Section 3.4 we will describe this in more detail. Our search mechanism ensures the following:

1. Always a token is selected whose owner is as close as possible to the joining node.
2. If there are several tokens whose owners are close to the joining node, the one with the lowest height is chosen.

In summary, if a requested token is available locally at the requestor's home repServer, the requestor and the owner of this token are in the same domain. This is the best case in terms of communication overhead between the repServers and communication distance between requestor and owner. If a token is not available locally, the ERS search procedure tries to find a token in a domain close to the requestor's domain.

3.3 Token information

A repServer stores all tokens of a group in a token basket, i.e. one token basket exists for each group known at the repServer. A token basket has the following structure:

- *Group*: Unique multicast group identifier.
- *SetOfTokenPackets*: The tokens are grouped according to their owner into so-called token packets.

Each token packet includes the following information:

- *Owner*: Unique identifier of the tokens' owner. A node receiving a token from this packet is allowed to connect to owner in the corresponding ACK tree.
- *Height*: Owner's height in the corresponding ACK tree. The height is used to distinguish the "quality" of alternative tokens. A token with low height is preferable since its use results in an ACK tree with low height and therefore low average path length.
- *Tokens*: Amount of tokens in this token packet.
- *ExpDate*: Expiration date of the token packet (see Section 4).

A token basket is to be established when the first set of tokens associated with the corresponding group is created and it is deleted when the last of this group's token has been removed from it. Each token basket contains a set of token packets. A token packet encloses all of a group's tokens belonging to the same owner.

3.4 Group management operations

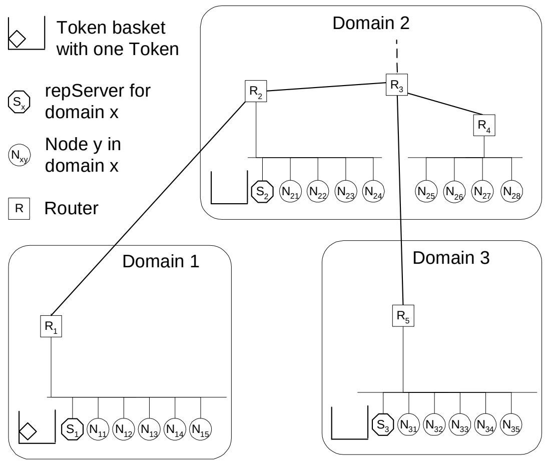
In this section we will describe the group management operations create group, delete group, join group and leave group in more detail. When describing these operations, we assume the absence of failures. Communication and node failures will be considered in Section 4.

All descriptions in this section refer only to the TRS that creates the ACK tree and not the multicast transport protocol that uses the ACK tree. It is necessary to explicitly distinguish between those two protocol classes since the multicast transport protocol has to implement the same group management operations. For a description of ACK tree based multicast transport protocols see [3, 4, 5].

A node *N* creates a new multicast group by initiating a *repCreateGroup (Group, K)* operation at its home repServer *S*. Subsequently, *S* creates a token basket for *Group* including one token packet with owner *N*. *K* specifies the amount of tokens in the token packet. The height of the token packet is initialized with one, because owner *N* as the root node has the height one in the ACK tree. For example assume node N_{11} in Figure 1 sends a *repCreateGroup* operation to its home repServer S_1, responsible for domain 1. Subsequently S_1 will create a token basket for this group. Figure 2 depicts this scenario in more detail and after the token basket is created.

When the operation *repDeleteGroup (Group)* is invoked at a repServer, this server deletes the *Group*'s token basket and sends a *DeleteGroup (Group)* message to all other repServers. Since all repServers belong to a well-known multicast group, this can simply be done by a multicast message. Each repServer receiving *DeleteGroup (Group)* removes the *Group*'s token basket. Note that it is sufficient to send *DeleteGroup* by the best effort IP multicast service since the expiration date mechanism described in Section 4 ensures that all outdated state information is removed despite of node and communication failures.

When a node triggers a *repJoinGroup (Group, NewMember, K)* operation at its home repServer *S*, *S* checks whether a token for *Group* is locally available. If such a token exists locally, *S* removes one token with lowest height from the token packet and sends it to the requestor of *repJoinGroup*. Subsequently, *S* creates a new token packet for owner *NewMember* with *K* tokens. The height of the new token packet is the height of the delivered token increased by one. Assume for example that in the scenario depicted in Figure



Token basket with one Token
repServer for domain x
Node y in domain x
Router
Domain 1
Domain 2
Domain 3
R_1
R_2
R_3
R_4
R_5
S_1
S_2
S_3
N_{11} N_{12} N_{13} N_{14} N_{15}
N_{21} N_{22} N_{23} N_{24} N_{25} N_{26} N_{27} N_{28}
N_{31} N_{32} N_{33} N_{34} N_{35}

Fig. 2. Example scenario

2 a node belonging to domain 1, say N_{12}, sends a *repJoinGroup* operation to its home repServer S_1. S_1 has a locally available token for the requested group which is delivered to N_{12}.

Now we will consider the situation that a repServer *S* has no local tokens for a requested group. If this is the case, *S* initiates a token search by using ERS. The search starts with a multicast *TokenSearch* message to the repServers' group address with a TTL of one. If no repServer returns an answer within a certain time, *S* repeats the search message with an increased TTL, and again waits for an answer. This process is continued until *S* receives an answer or the maximum TTL of 255 is reached.

A repServer receiving a *TokenSearch (Group, Requestor)* message has to check whether it has a token for the requested *Group*. If this is the case, it responds to *Requestor* with an unicasted *TokenAvail (Group, Height, Provider)* message, where *Height* is the minimal height of *Group*'s local tokens at the token *Provider.* Since each ERS search message may result in more than one answer, *S* has to choose one responding node. The *Height* value is used as a token quality metric. *S* chooses the responding token provider *R* with the lowest token height by sending a unicast *GetToken* message to *R*. Finally *S* receives the requested token with a *Token* message from *R* and *R* removes this token from its token packet.

After *S* has received a token, it establishes a token basket for *Group*, including a token packet for *NewMember,* where *NewMember* was the caller of *repJoinGroup*. Then the received token is handed over to *NewMember.*

To illustrate the token search procedure by means of an example, take Figure 2 and assume that node N_{31} wants to join a group and therefore sends a *repJoinGroup* message to its repServer S_3. S_3 checks whether it has a token for this group. Since there is no locally available token, S_3 has to initiate a token search by multicasting *TokenSearch* with increasing TTL until a token is found. The first few multicast messages with a TTL less than 4 do not reach other repServers. The token search message with a TTL of 4 is received by repServer S_2, however S_2 has no tokens and therefore does not reply to S_3. The next search message with a TTL of 5 is received by S_1, which owns a suitable token and answers with a *TokenAvail* message. Subsequently, S_3 stops multicasting *TokenSearch* messages and sends a *GetToken* message to S_1. Finally, S_1 sends the requested token to S_3 which forwards it to the searching node N_{31}. Since S_3 creates new tokens with owner N_{31}, following join requests in domain 3 can be processed by S_3 without further token searches. The entire message sequence for this example scenario is depicted in Figure 3.

During this two phase token search procedure - phase one includes *TokenSearch* and *TokenAvail*, phase two includes *GetToken* and *Token* messages - the following infrequent situation may occur. If a repServer responds with a *TokenAvail* message, it indicates that at this moment a suitable token is available, i.e. the token will not be reserved for the requesting repServer. Note that this design leads to a stateless and thus light-weight protocol. For example assume that S_1 in Figure 2 has only one token and receives two *TokenSearch* messages, one from S_2 at time t_2 and one from S_3 at time t_3, where t_2 is before t_3. S_1 responds to S_2 with *TokenAvail* but also to S_3 since S_1 cannot know whether S_2 will choose S_1's token or has already chosen another one. Therefore, it can occur that both, S_2 and S_3 request S_1's token by sending a *GetToken* message. In this case, S_1 hands over its token to the first caller of *GetToken*; all other requestors receives a *NoToken* message, instead.

If a repServer *S* receives a *NoToken* message it simply chooses another repServer provided that *S* has received more than one *TokenAvail* message in the first search phase. Otherwise, *S* simply continues the token search procedure by sending a new *TokenSearch* message with increased TTL. The internal messages for the token search procedure are summarized in Table 2.

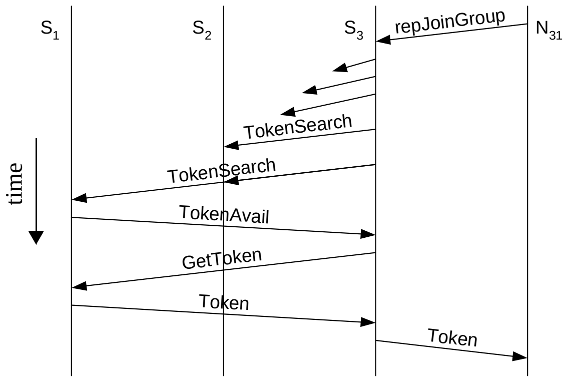


Fig. 3. A typical token search

Table 2. Internal TRS-PS operations

Operation	Description
TokenSearch (Group, Requestor)	Multicasted by a repServer identified by *Requestor* to search for a token of *Group.*
TokenAvail (Group, Height, Provider)	*TokenAvail* is the response to *TokenSearch* if the *Provider* has a suitable token with height given by *Height.*
GetToken (Group, Requestor)	Sent by the token searching *Requestor* to the *Provider* of *TokenAvail* to get a token for *Group*.
Token (Group, Owner, Height)	*Token* is sent by the *Provider* as a reply to the *GetToken* request if a token for *Group* is available.
NoToken (Group)	Sent by the *Provider* as a reply to the *GetToken* request if no token for *Group* is available.

When a node N leaves a group, all of its tokens are removed. The used multicast transport protocol must ensure that a node is only allowed to leave a group if it has no child nodes in the ACK tree, i.e. is a leaf node. A non-leaf node can leave a group after it has arranged a rejoining for all child nodes at other ACK tree nodes. As we assume that a node has no descendants in the ACK tree when it leaves the group, all tokens owned by N are in the group's token basket stored on N's home repServer S. When receiving repLeaveGroup (Group, Member), S removes N's token packet from the Group's token basket.

If N leaves a group this affects not only the tokens owned by N, but also the token owned by N's parent in the ACK tree. Conceptually, if N leaves a group it releases its parent's token allocated by N so far. Hence N's parent adds this token by means of the repAddToken operation to the token basket of its home repServer when it recognizes that N leaves the group.

4. Protocol in the presence of failures

In the previous section we have described the group management operations during normal conditions without considering communication and node failures. In this section we will describe the behavior in such failure situations.

All token information is maintained according to the soft state principle [12]. Token packets are associated with an expiration date. If the expiration date is reached, it must either be extended or the token packet will be discarded automatically. Obviously, the lifetime of a token packet depends on the lifetime of its owner. To prevent a token packet from expiring, the token's owner periodically refreshes the token information of not already used tokens, which extends their expiration date. If no refresh message is received within two refresh cycles, the token packet is discarded. On the other hand, if a refresh message is received without storing the corresponding tokens, these tokens are created.

Although our protocols discard token packets explicitly during normal operations, this mechanism allows to design a robust but nevertheless light-weight protocol, ensuring even in the presence of node and communication failures that eventually all outdated information is removed. In addition, this mechanism allows us to keep token information in volatile memory which is necessary to provide a high repServer throughput. If the token information is lost due to a repServer crash, the refresh mechanism recovers the lost data. In summary, if a node is alive and not occupied, the network is not partitioned and the home repServer is also alive, tokens of these node are maintained by the TRS. This behavior is known as the worthwhile fate-sharing design principle [12].

As this mechanism is only necessary to discard outdated information and recover tokens in case of node or communication failures, the refresh cycles can be rather large. Furthermore, only nodes that are not already occupied need to refresh their token information, therefore, the communication overhead is low.

If a repServer has restarted and must reply to a repJoinGroup message before the token information is recovered, it simply starts a token search procedure. If the home repServer is not available when *repCreateGroup*, *repDeleteGroup* or *repJoinGroup* is to be performed, some other repServer can be selected to execute those operations. Of course, when selecting another repServer, those that are in close domains are preferable. To be able to select another repServer, each node can maintain a list of repServers. The first and most preferable repServer in this list is the home repServer in this node's domain. The following elements in the list enumerate all alternative repServers with decreasing suitability.

In case of an unavailable home repServer, the caller of *repLeaveGroup* can give up as the expiration date mechanism will ensure that the corresponding token packet will be deleted. Of course, it makes no sense to select another repServer because the tokens can be deleted only at the repServer, the corresponding *repJoinGroup* operation was performed at.

Alternatively, if the home repServer is not available, the sender of *repCreateGroup* and *repDeleteGroup* can give up, too. In the case of *repCreateGroup* no token information will be established, which is treated the same way as the loss of tokens due to crashes (see above). In the case of *repDeleteGroup* the expiration date mechanism will eventually remove all outdated tokens.

In summary, the token repository service is still operational even in failure situation. Therefore, the proposed mechanism is as robust as ERS.

5. Performance evaluation

In this section we present some analysis and simulations comparing the TRS approach with ERS and ERA. The analytical studies evaluate the maximum message overhead and the height of the created ACK tree. The simulations additionally evaluate the ACK tree delay.

5.1 Analysis

5.1.1 Message overhead. The following message overhead evaluation considers only the overhead for group management rather than the overhead on routing layer or the reliable multicast transport protocol. We assume a scenario in which the join and leave operations are independent, i.e. all join operations are processed before the first leave operation and we do not consider possible rejoining overhead when non-leaf nodes leave the ACK tree. Furthermore we assume the absence of failures. As the message overhead of ERA depends mainly on the time period, it is not considered here.

Using ERS, create, delete and leave a group is not explicitly done, therefore the message overhead is 0. The worst case for joining a group is that 255 multicast search messages must be sent to find a parent node, since 255 is the maximum time-to-live value in an IP packet and that all nodes that have already joined the ACK tree reply to the searching node. The maximum number of messages m_j for joining a group is therefore as follows:

$$\begin{aligned} m_j &= \text{search messages} + \text{reply messages} \\ &= 255_m\, j + \sum_{i=1}^{j} i_u \\ &= 255_m\, j + \frac{j_u^2 + j_u}{2} \end{aligned} \tag{1}$$

Index u/m identifies unicast/multicast messages and j is the number of join operations (see Table 3). Since there is a square component in the formula, the worst case message overhead is quadratic.

The message overhead for creating a group using TRS is one message, the repCreateGroup message. To delete a group, repDeleteGroup is sent to the home repServer which sends one multicast DeleteGroup message.

To join a group repJoinGroup must be sent to the repServer and then a token is replied. If a repServer has no local token for a requested group, a token search is invoked by sending multicast search messages. In the worst case 255 search messages are sent and every other repServer sends an answer message. Finally, the token is handed over, which needs additionally two messages. Such a token search is processed only once per repServer and the repServer at which the group is created needs no token search at all. The maximum number of messages for joining a group is therefore as follows, where B is the number of requested repServers:

$$\begin{aligned} m_j &= 2_u\, j + (B-1)(255_m + (B-1)_u + 2_u) \\ &= 2_u\, j + 255(B-1)_m + B_u^2 - 1_u \end{aligned} \tag{2}$$

If we assume that we have a large number of join operations, that means $j >> B$ then:

$$m_j \approx 2_u\, j \tag{3}$$

This means, the number of messages rises linear with the number of join operations. To leave a group only one message, repLeaveGroup is sent to the repServer.

The following numerical example illustrates the results. Assume that a multicast groups is created, 1000 nodes join the group, 500 nodes leave the group and 100 repServers are involved. ERS results in a maximum overhead of more than 755 thousand messages while TRS results in less than 38 thousand messages.

5.1.2 Tree height. Using ERS/ERA, the maximum height of the created ACK tree is only limited by the number of join operations, i.e. in the worst case the height can be equal to *j-l+1*, where *j* is the number of join operations and *l* the number of leave operations. With TRS the height of the created ACK tree is determined by the number of join operations and the number of requested repServers. If only one repServer is requested, a tree with minimal height is created since for each token request the token with minimal height in the ACK tree is delivered. Therefore the height can be calculated as follows:

$$\text{Amount of nodes in a complete k-ary tree:}$$

$$N = \sum_{i=0}^{h-1} k^i = k^0 + k^1 + \ldots + k^{h-2} + k^{h-1}$$

$$N = \frac{(1-k)k^0}{(1-k)} + \frac{(1-k)k^1}{(1-k)} + \ldots + \frac{(1-k)k^{h-2}}{(1-k)} + \frac{(1-k)k^{h-1}}{(1-k)}$$

$$N = \frac{k^0 - k^1 + k^1 - k^2 + \ldots + k^{h-2} - k^{h-1} + k^{h-1} - k^h}{(1-k)}$$

$$N = \frac{1-k^h}{(1-k)} \quad \Rightarrow \quad h = \lceil \log_k (N(k-1)+1) \rceil$$

$$\text{Tree height with j join operations:}$$

$$h = \lceil \log_k ((j+1)(k-1)+1) \rceil \tag{4}$$

If we consider *B* instead of one repServer, the worst case is that at *B-1* repServers only one join operation is processed and that each of these join operations results in a parent with maximum height in the ACK tree. All other join operations are processed by one repServer. The maximum height can be expressed as follows:

$$h = B - 1 + \textit{height of a complete } k - \textit{ary tree}$$
$$\textit{for } (j - (B-1)) \textit{ join operations}$$
$$= B - 1 + \lceil \log_k ((j - B + 2)(k-1)+1) \rceil \tag{5}$$

Again we want to make a numerical example. We assume that 1000 join operations and 500 leave operations are made and that 100 repServers are involved. The ACK tree is k-bounded with *k*=10. While ERS can result in an ACK tree of height 501, TRS guarantees a maximum height of 103.

5.2 Simulations

We have performed simulations to compare TRS with ERS and ERA. Before the simulation results are presented, the simulated scenario is described.

5.2.1 Simulation scenario. Our simulations are performed using the NS2 [13] network simulator. Our simulation scenarios consist of various hierarchical networks with 100 to almost 2000 nodes, including LAN, MAN and WAN structures. The networks were generated with Tiers [14]. All links are symmetrical duplex links with the bandwidths 10Mbps for LANs, 100Mbps for MANs and WANs. The link delays are chosen randomly for each link from 1ms to 3ms for LANs, 1ms to 8ms for MANs and 5ms to 19ms for WANs.

Since the multicast routing protocol significantly influences the measured results, we have run most simulations with both, the distance vector multicast routing protocol (DVMRP) [8] and protocol independent multicast sparse mode (PIM-SM) [9]. All routers in the network use drop tail queues.

Table 3. Notation and summary of analytical results

m_x............	Number of messages to create (x=c), delete (x=d), join (x=j) or leave (x=l) a group.
c, d, j, l	Number of create, delete, join or leave operations.
X_u, X_m......	X unicast or multicast messages.
B...............	Number of repServers requested for a token.
N...............	Amount of nodes in a complete k-ary tree.
h	Height of the created ACK tree.

Maximum message overhead with ERS:

$$m_c = 0, \quad m_d = 0, \quad m_j = 255_m j + \frac{j_u^2 + j_u}{2}, \quad m_l = 0$$

Maximum message overhead with TRS:

$$m_c = c_u, \quad m_d = d_u + d_m$$
$$m_j = 2_u j + 255(B-1)_m + B_u^2 - 1_u \approx 2_u j, \quad j >> B$$
$$m_l = l_u$$

Maximum tree height with ERS/ERA:

$$h = j - l + 1$$

Maximum tree height with TRS:

$$h = B - 1 + \lceil \log_k ((j - B + 2)(k-1)+1) \rceil$$

In the simulations we investigate the behavior of the various protocols during normal conditions, i.e. we do not consider error situations. All messages were delivered reliably, nodes do not fail and the network is not partitioned.

Some simulations are performed with various background traffic conditions. The exponentially distributed background traffic is generated by randomly placed senders and receivers of TCP streams. 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 all bandwidths by factor 100 in order to run the simulations with background traffic.

The token repository service is configured with 8 repServers. ERS is configured with various timeout intervals which are described in the next section. ERA is also configured with various timeout intervals and with a fixed TTL scope of 127 as it is used in RMTP [4]. In all simulations the ACK tree is bounded with *k* = 10.

5.2.2 Simulation results. Our first simulation result in Fig. 4 depicts the dependency between received messages on transport layer and various levels of background load. Using the ERS or ERA approach, messages are received by the group members, while in the TRS approach messages are additionally received by the repServers. In this simulation study we have used 200 join operations and the routing protocol DVMRP. The background load is measured as the percentage of busy links during the simulation time. A load of 100% means that each network link was busy, i.e. has a non-empty outgoing routing queue, during the entire simulation.

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

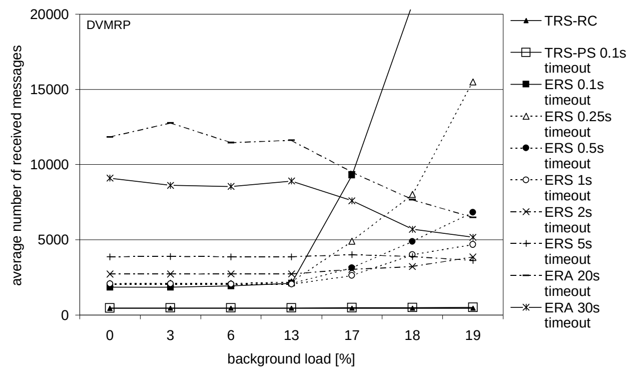


Fig. 4. Messages received depending on the background load

timeout interval, i.e. the time per hop, a node waits for an answer to arrive, the sender of ERS starts a new multicast search message with increased TTL. For example, if the timeout is one second and the search scope ten hops then the node performing ERS waits ten seconds for an answer before it starts a new search. The lower this time to wait is, the sooner a new search message is sent and therefore the earlier the effect of exponentially increasing message overhead occurs. However, the timeout parameter can only be increased within a certain range, since this influences 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 case of low background load. For example, in Fig. 4 the message overhead of ERS with 5s timeout interval, which is up to 13% background load higher than that of the other ERS curves with lower timeout intervals. Since it takes longer for a node to join the ACK tree if the timeout interval is increased, it also takes longer before the joining node itself is able to accept child nodes. Therefore, other joining nodes must possibly search in a larger scope to connect to the ACK tree.

ERA results in a high message overhead independent of the background load. 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 network congestion and consequently to a high message delay. Since the simulated time period was restricted, not all invitation messages were delivered during simulation time.

The message overhead of both TRS strategies, TRS-RC and TRS-PS is much lower compared to ERS and ERA and moreover, independent of the background load, always constant. Both strategies result in about the same message overhead. We have also simulated the proxy server strategy with various timeout intervals but the results have differed only slightly.

We have made another simulation study to investigate the impact of network size. Fig. 5 shows the results for 50 join operations with DVMRP and PIM-SM routing. The message overhead of TRS-RC and ERA is constant, independent of the network size and routing protocol. The results show that TRS-PS sends more messages than TRS-RC and the number of messages increases with larger networks. However, compared to ERS and ERA the number of messages is always smaller and rises only slightly with the network size.

Fig. 6 shows the round trip delay of the various approaches depending on the number of join operations in a network with 251 nodes. The round trip delay is the time between sending a message and receiving the last aggregated ACK at the root node.

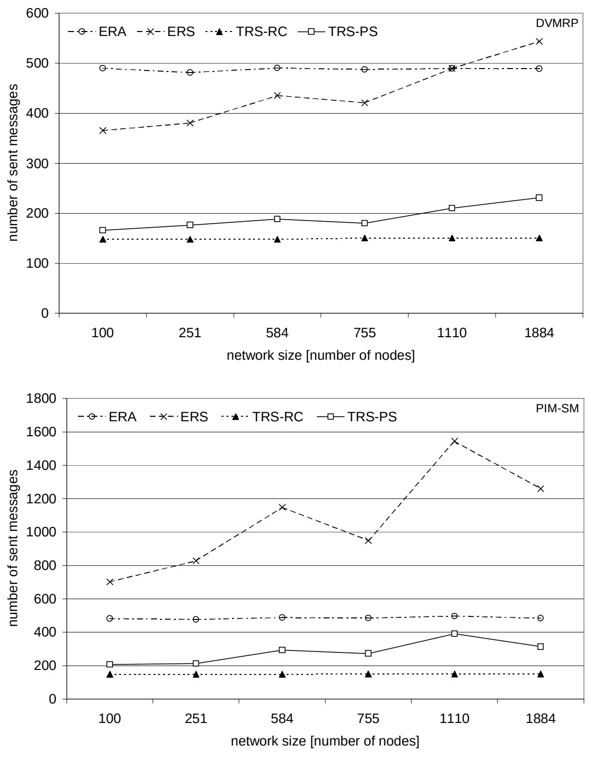


Fig. 5. Messages sent depending on the network size

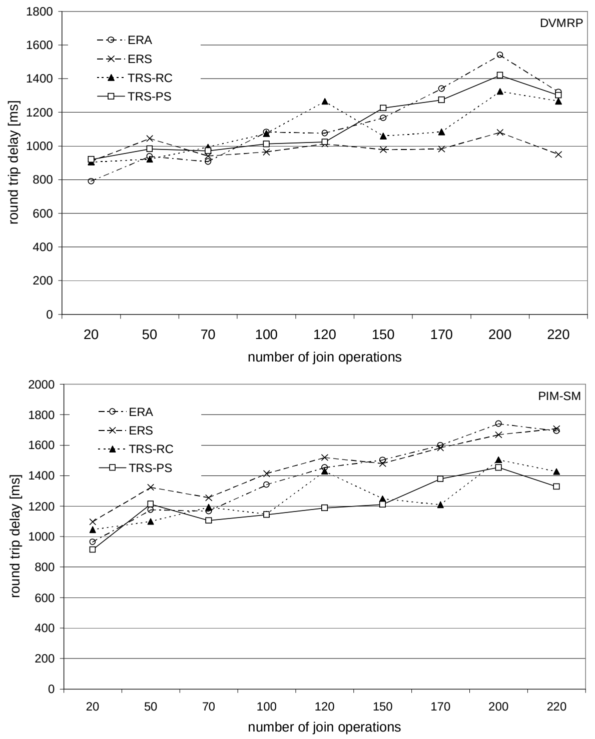


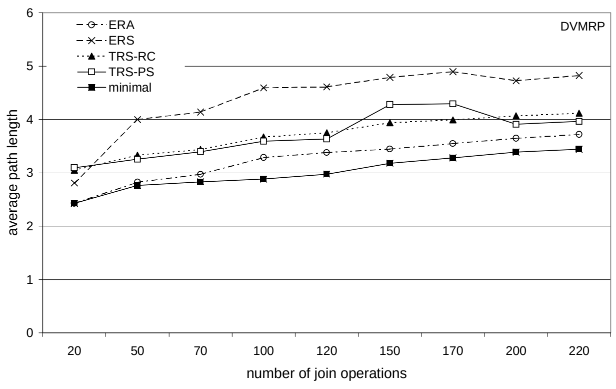
Fig. 6. Round trip delay depending on the number of joins

As it can be seen in the chart, TRS-PS results in a low round-trip delay. Independent of the routing protocol TRS-PS performs better than ERA and with PIM-SM routing even better than ERS. Low delays are desirable since the average throughput of a reliable (multicast) channel with a window-based sending scheme is limited by the quotient of buffer size / round trip delay. The lower the delay is the higher is the throughput respectively the lower is the required buffer size.

Above all, in the case of PIM-SM routing the proxy server approach is an appreciable improvement compared to ERS and ERA. The poor results of ERS and ERA with PIM-SM originate from the dissemination of multicast messages starting always from the same core node in the network for all senders. Therefore, ERS finds always nodes close to this core rather than close to the searching node which results in high round trip delays. The use of ERA leads to a similar problem, since the distance between the invitation sending node and the receiver cannot be determined by analyzing the TTL values which will only provide the distance between the receiver and the core.

The last simulation results depicted in Fig. 7 investigate the average path length of the created ACK trees. The path length affects the reliability of the created ACK tree. The multicast service may be disrupted for a node if one of its parents in the ACK tree becomes unavailable. Therefore, the lower the number of parents the higher the reliability from this node's perspective. So, the average path length of the ACK tree can be used as a quality criterion for reliability, since it is equivalent to the average number of nodes that must rejoin the tree if a single ACK tree node fails.

Fig. 7 shows that TRS-PS as well as TRS-RC and ERA lead to ACK trees with low path lengths that are near to the theoretical minimum. The use of ERS results in unbalanced ACK trees especially in combination with the routing protocol PIM-SM, i.e. the failure of a single node may lead to a vast overhead for example for rejoining its child nodes.



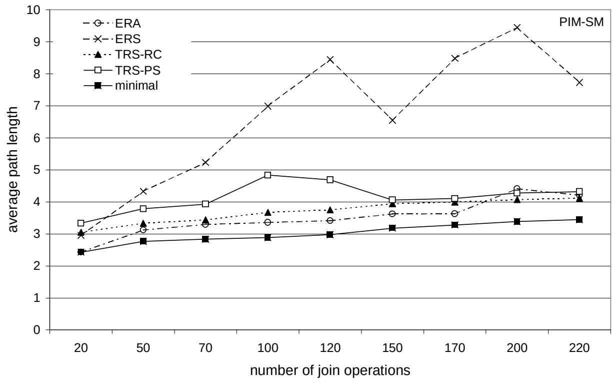


Fig. 7. Average path length of the created ACK tree

In summary, the presented simulations illustrate that TRS-PS performs better than ERS and ERA in terms of message overhead and scalability and in many cases better in terms of round-trip delay and reliability.

6. Summary

The presented token repository service with proxy server strategy (TRS-PS) is an efficient and robust approach for constructing ACK trees. Compared to the already proposed random-choice strategy, the proxy server strategy is a simpler strategy that needs to maintain less state information and therefore is easier to implement. Compared to the various approaches based on expanding ring search, TRS-PS has several advantages like improved scalability, the usability even for unidirectional multicast networks, the generation of well-shaped ACK trees with low round trip delay and high reliability independent of the multicast routing protocol.

7. References

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

[2] 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, pp. 112-121.

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

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

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

[6] 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, pp. 365-376.

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

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

[9] 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.

[10] Maihöfer, C.: Improving multicast ACK tree construction with the Token Repository Service, Proceedings of the 2000 IEEE ICDCS Workshop on Group Computation and Communication, 2000, pp. C57-C64.

[11] Rothermel, K.; Maihöfer, C.: A robust and efficient mechanism for constructing multicast acknowledgment trees, Proceedings of the IEEE Eight International Conference on Computer Communications and Networks (IEEE IC-CCN'99), 1999, pp. 139-145.

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

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

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