THE TOKEN REPOSITORY SERVICE

A Universal and Scalable Mechanism for Constructing Multicast Acknowledgment Trees

Christian Maihöfer

University of Stuttgart
Institute of Parallel and Distributed High-Performance Systems (IPVR)
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany
maihoefer@informatik.uni-stuttgart.de

Abstract
Many new applications like groupware systems, news and file distribution or audio and video systems are based on multicast as a prerequisite for scalability. Many of these applications need a reliable multicast support, which is realized in a scalable way by tree based multicast transport protocols, where the receivers are organized in a so-called ACK tree. Tree based approaches raise the problem of setting up and maintaining the ACK tree, which is usually done by variations of the expanding ring search (ERS) approach. In this paper we present an alternative approach for building up ACK trees that is based on the concept of a distributed token repository service. Our analysis and simulations show that our approach leads to a significantly lower message overhead compared to ERS and results in better shaped ACK trees, which has a positive effect on reliability and delay.

Keywords
reliable multicast, tree-based, TRS, ERS, ACK tree, analysis, simulation

1. INTRODUCTION

Multicast support is a prerequisite for many applications to ensure scalability for large receiver groups. Although multicast support is already available in the Internet, the provided IP multicast service offers only best effort semantics [5]. Several protocols have been proposed to overcome this drawback by a protocol layer on top of IP multicast [7, 10, 16, 19, 20].

All reliable multicast protocols are based on the same concept, controlling the successful delivery by some kind of acknowledgments returned by the receivers to the source. Simple approaches, where all receivers send their acknowledgment messages directly to the sender can cause the well-known ACK implosion problem [9, 13, 15]. To overcome the ACK implosion problem, the most promising approaches are tree-based protocols [4, 7, 10, 19, 20]. They ensure scalability by organizing all group members in a so-called ACK tree. Instead of sending an acknowledgment message directly to the sender, each receiver confirms the correct delivery only to its parent in the ACK tree, which is responsible for possible retransmits. Dependent on the concrete protocol, an inner node in the ACK tree sends an acknowledgment to its parent either after it has received the corresponding message correctly or it first collects all acknowledgments from child nodes. This means, in the latter case an inner node sends an aggregated ACK to its parent after it has received the multicast message and the corresponding ACK from each child, confirming the correct message delivery for the entire subhierarchy. Since each node in the ACK tree has an upper bound on the number of its children, no node and no part of the network is congested with messages.

Tree-based protocols raise the problem of setting up 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). 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. However, our performance evaluations will show that the use of ERS on a large scale has several shortcomings, since it results in a large message overhead and causes particular problems in combination with source-based or unidirectional core based routed networks.

In this paper we propose an alternative approach for constructing ACK trees, called *token repository service* (TRS). Our approach is based on a distributed token repository. 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.

We have developed three strategies to implement the TRS, the proxy-server strategy (TRS-PS) [11, 12], the random-choice strategy (TRS-RC) [17] and the minimal-height strategy (TRS-MH) [14]. The strategies have different characteristics. The proxy-server strategy is easy to implement and to integrate into the Internet structure. The random-choice strategy results in better shaped ACK trees and lower message overhead, but on the other hand needs an infrastructure to be established. The minimal-height strategy is an extension to the random-choice strategy, creating ACK trees with minimal height. In contrast to ERS, all three strategies provide scalability in the ACK tree construction and better shaped ACK trees, necessary for ensuring reliable multicasting with high reliability, low delays and high throughput.

The remainder of this paper is organized as follows. In the next section the background and related work are discussed. Section 3 gives an overview of the token repository service. In the following sections, the three TRS strategies are described in detail. The behaviour of our approach in the presence of failures is considered in Section 7. In Section 8 and Section 9 performance evaluations based on theoretical analysis and simulations are presented before we conclude with a brief summary.

2. BACKGROUND AND RELATED WORK

When a new member joins a reliable multicast group the question arises how it will be connected to the group's ACK tree. The problem is to connect the new member to a *k*-bounded parent that is not already *occupied*, i.e. has not already *k* children. *k* is the maximum number of children a node can accept. The bound *k* for a node depends on various characteristics, such as the node's performance, reliability or load.

Most approaches to establish an ACK tree are based on expanding ring search (ERS) [20]. ERS is a common technique to search for resources in a network [2]. 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 (see Figure 1). 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.

Several proposed protocols reverse the method described above by making the non-occupied ACK tree nodes search for child nodes with multicast invitation messages (ERA, expanding ring advertisement) [10, 7] and some protocols use a combination of both approaches [4].

Our analysis and simulation results will show that ERS/ERA result in a large message overhead. An additional drawback of ERS and ERA are their dependency on the various routing protocols, resulting in particular problems with each of them. ERS and ERA with distance vector multicast routing (DVMRP) [18] lead to a vast overhead at all involved routers because a new multicast routing tree is to be build for each sender. This means each node that joins a group via ERS enforces a new, separate routing tree. If ERA is used, a routing tree must be maintained for all non-occupied nodes in the ACK tree. Note that if a member is only a receiver of multicast messages, these trees are only used for the ERS/ERA search. With an unidirectional shared tree approach like PIM-SM [6], the use of ERS and

ERA result in a traffic concentration at the core, an even higher message overhead compared to DVMRP and ACK trees of poor quality with respect to tree height and delay.

A further serious drawback of ERA is the message overhead due to the invitation messages, which are sent even if no node wants to join. ERS has the additional drawback that it cannot be used in unidirectional multicast networks. For example, ERS cannot be used in satellite broadcast networks, where there is no multicast backchannel or only an inefficient one.

3. OVERVIEW OF THE TOKEN REPOSITORY SERVICE

In this section we will describe the interface, concept and implementation idea of the token repository service, which is our proposed infrastructure for building up ACK trees. Then in the following sections 4, 5 and 6 the three strategies TRS-PS, TRS-RC and TRS-MH are described in detail.

3.1 Interface and Concept of the Token Repository Service

The basic concept of our approach are tokens which represent the right to connect to a certain 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

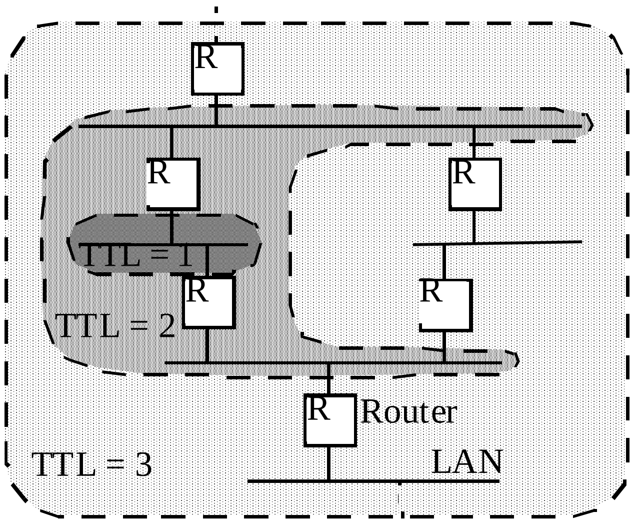


Figure 1: Increasing search radius of ERS

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 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.

Initially, there are *k* tokens of a group in the repository, generated on behalf of a create group operation. When a node, say *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, returns it to *N* and generates new tokens with owner *N*. 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. The operations provided by the TRS are summarized in Table 1.

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, New-Member, K) returns (Token)	*repJoinGroup* is called when the node identified by *NewMember* wants to join *Group*, where *NewMember* is *K*- bounded. 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 to the repository owned by *Owner*. It is called by *Owner* when a child of *Owner* disconnects from the *Group*'s ACK tree.
repRefreshToken (Group, Owner, Number)	To provide fault tolerance, *repRefreshToken* is periodically called by the tokens' owner. It indicates how many child nodes (*Number*) can still be accepted (see Section 7).

3.2 Implementation of the Token Repository Service

In this section, we will describe the basic principles of implementing the TRS. 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 domain, where each domain encompasses a disjunct set of nodes. For example, repServer S_1 in Figure 2 is responsible for domain 1 consisting of nodes N_{1x}. Domains should structure the network by communication distance, i.e. the communication distance between two nodes in the same domain is typically smaller than between two nodes in different domains. A repServer, responsible for a particular node in its domain, is called this node's *home repServ*er. In Figure 2, S_1 is the home repServer of the nodes N_{1x}. During normal operation nodes access the token repository service only via their home repServer (see Section 7 for failure situations).

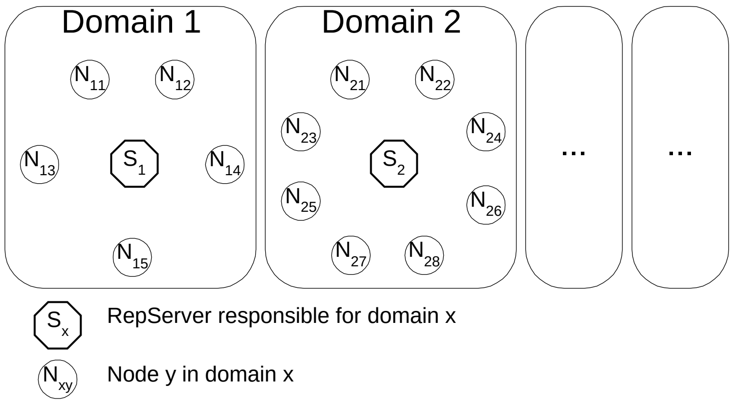


Figure 2: Domain structure

Tokens are stored on the distributed repServers. Note that not all tokens of a group are stored at only one repServer, thus several repServers may store tokens for the same group and usually a repServer stores tokens of a number of groups.

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. Since a token is always stored at the repServer responsible for its owner's domain, a repServer possesses tokens of a group only when a node in its domain has created or joined this group before. As a consequence, usually 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, 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. In the proxy-server strategy, a 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 the random-choice and minimal-height strategy, all repServers are organized in a tree structure. Nodes access the token repository service only via leaf repServers, which store the token information. Non-leaf repServers are necessary to facilitate the token search procedure, if no local token is available. Each non-leaf repServer maintains a group-specific set of all child repServers that belong to a token containing subhierarchy. A token search is processed by forwarding the search step-by-step to the parent in the repServer hierarchy, i.e. in leaf-to-root direction until one is reached which knows a token containing subhierarchy. Then the search is forwarded in reverse direction, i.e. in root-to-leaf direction to such a child repServer. This is repeated until a leaf repServer is reached which hands over a token to the searching node.

Note that our search mechanism ensures the following:

1. always a token is selected whose owner is as close as possible to the joining node and
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 in the ACK tree. If a token is not available locally, the 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.
- NoOfTokens: Number of tokens in this token packet.
- ExpDate*:* Expiration date of the token packet (see Section 7).

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.

4. THE TOKEN REPOSITORY SERVICE WITH PROXY SERVER STRATEGY (TRS-PS)

In this section we will describe the group management operations create group, join group, leave group and delete group for TRS-PS in more detail. TRS-PS is the first implementation strategy, which is based on ERS. When describing these operations, we assume the absence of failures. Communication and node failures will be considered in Section 7.

4.1 Create Group Operation

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

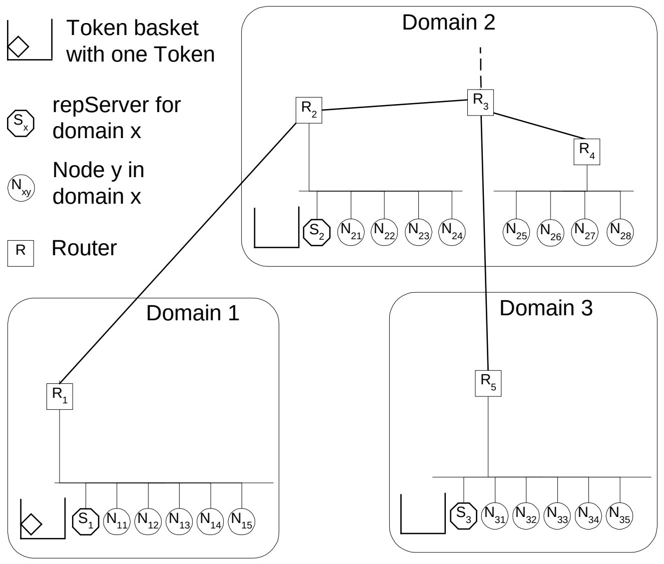


Figure 3: Example scenario

domain 1. Subsequently, S_1 will create a token basket for this group. Figure 3 depicts this scenario in more detail and after the token basket is created.

4.2 Delete Group Operation

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 7 ensures that all outdated state information is finally removed despite of node and communication failures.

4.3 Join Group Operation

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 3 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 (Group, Requestor)* 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 3 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.

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 3 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.

4.4 Leave Group Operation

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 children 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 (see Table 1).

Note that this mechanism, namely adding tokens by the parent, ensures robustness of our approach. Assume that a node in the ACK tree crashes. The crashed node is not able to return its allocated token to the TRS. Therefore, the parent of the leaving or crashed node has to add the token.

5. THE TOKEN REPOSITORY SERVICE WITH RANDOM CHOICE STRATEGY (TRS-RC)

For the TRS-RC strategy, the network is also structured into domains but in contrast to TRS-PS, the domains are hierarchical. The root domain includes all nodes of the network, while the leaf domains encompass disjunct set of nodes. Inner domains contain the nodes of their child domains. Figure 4 shows the association of nodes to domains. RepServer S_1 is responsible for domain 1 and repServer S_4 is responsible for domain 4, which consists of several subdomains.

A client accesses the token repository service only via the leaf repServers

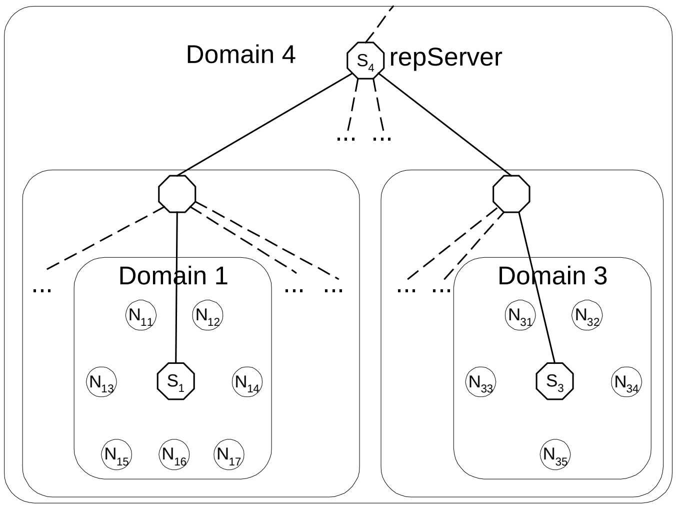


Figure 4: Hierarchical domain structure of TRS-RC and TRS-MH

in its domain, called this node's *home repServer*. In Figure 4, S_1 is the home repServer for all nodes N_{1x} of domain 1.

All token information is stored on leaf repServers. 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 transitive parents of these nodes in the repServer hierarchy. 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 transitive (leaf) children store tokens of *G*. A group record of repServer *S* includes the following fields:

- Group: Identifier of the corresponding group.

- Sub*:* This is a bitmap encoding the list of *S*'s children in the group tree. Each entry in the bitmap corresponds with one child repServer. An entry is set equal to 1 if the corresponding child repServer is part of this group's group tree; otherwise it is set equal to 0.
- ExpDate*:* This field defines when the group record expires (see Section 7 for details).

Of course, a group tree may grow and shrink during its lifetime, i.e. group records are to be created, updated and deleted dynamically. 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 parent 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 or the root node is reached. 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 parent. A non-leaf repServer receiving this message checks whether the message's sender was its only child in the group tree. If this is the case, it deletes the group record and forwards the *NoTokenAvail* message to its parent. Otherwise, it just removes the message's sender from the group record's *Sub* list.

The following subsections describe the create group, delete group, join group and leave group operations in more detail.

5.1 Create Group Operation

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

Figure 5 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}, 1> in *G*'s token basket, assuming that N_{11} (i.e. the root of *G*'s ACK tree) is *k*-bounded, i.e. will accept at most *k* children. The height of the created tokens is 1, since N_{11} is the root node in the created ACK tree and we have defined the root node to has height 1 (see Section 3.1). Moreover, group records are established on each non-leaf

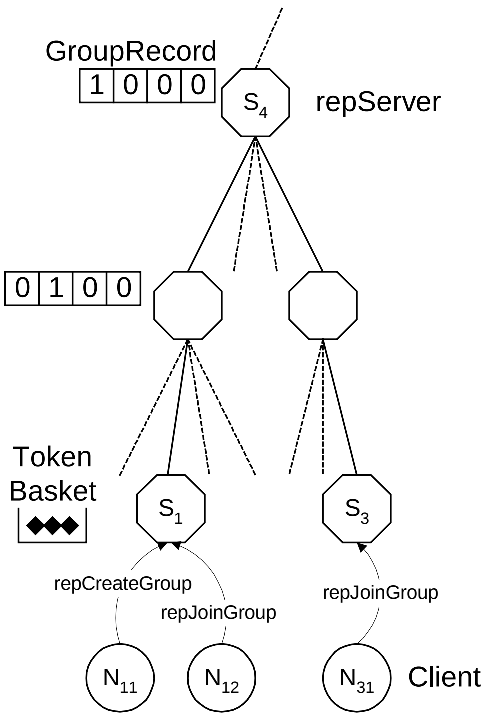


Figure 5: Create and join operations at the repServer hierarchy with TRS-RC

repServer along the path from S_1 to the root repServer. After their creation they indicate how to find a token containing leaf repServer, starting at the root node.

5.2 Delete Group Operation

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 parent. 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.

5.3 Join Group Operation

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 in the tokens' 3-tuple <group, owner, height> 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.

In Figure 5, 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 tokens for *G* are locally available. It finds a token <*G*, N_{11}, 1> and delivers it to N_{12}. In addition, it creates *k* tokens <*G*, N_{12}, 2> in *G*'s token basket, assuming that N_{12} will accept at most *k* children. After receiving token <*G*, N_{11}, 1>, N_{12} can connect to N_{11} in *G*'s ACK tree.

If no token basket exists for *Group* instead, *S* starts the token search procedure, which proceeds in two phases, the leaf-to-root directed and root-to-leaf directed search phase. *S* initiates the leaf-to-root directed search phase by sending a *TokenSearch* request to its parent 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 child 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 children in *Group*'s group tree. This request is forwarded along the edges of the group tree in a root-to-leaf direction until it arrives at a leaf repServer. If a repServer has more than one child 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, without using the TRS hierarchy, to *S*.

When receiving the token, *S* establishes a token basket, including 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 parent.

For example, assume that N_{31} in Figure 5 wants to join *G*. N_{31} 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 parent until the first repServer 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}, 1> at S_1 is selected and handed over to S_3. After that, S_3 delivers the token to N_{31}. Finally, group records for *G* are established on the path from S_4 to S_3, and new tokens <*G*, N_{31}, 2> 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.

5.4 Delete Group Operation

When a node, say *N*, leaves a group, it conceptually releases a token owned by its parent. Hence, *N*'s parent 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 children 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 to the parent repServer.

6. THE TOKEN REPOSITORY SERVICE WITH MINIMAL HEIGHT STRATEGY (TRS-MH)

In the following we will an overview of TRS-MH. The Minimal-Height strategy is described in detail, including pseudo code, in [14]. TRS-MH creates ACK trees with minimal height which improves reliability and round trip delay in the ACK tree. Note that the reliability of a node depends on its own reliability and the reliability of all nodes on the path from its parent node to the root node, because they are necessary to send a retransmission in case of data loss. Therefore, the lower the number of intermediate nodes on the path to the root, the higher the reliability from this node's perspective. Furthermore, small path lengths result usually in lower delays between the root node and all receivers, which improves throughput of protocols with aggregated acknowledgments [8, 19].

The basic concept of TRS-MH is quite simple. If a client requests a token from the token repository service, a token with minimal height in its 3-tuple <group, owner, height> has to be delivered. Therefore, new nodes connect always as close as possible to the root node in the ACK tree, which results in height-balanced ACK trees. A *height balanced* ACK tree is a tree with minimal height for a given *k*-bound, i.e. $\text{height}=\lceil \log_k(R(k-1)+1) \rceil$, where *R* is the number of nodes in the tree (see Section 8).

The implementation of TRS-MH is similar to TRS-RC but more complex than the previous strategies. Again, a group tree has to be maintained identifying the repServers that store tokens for a given group. Group tree information is stored in group records with the following structure:

- Group: Unique multicast group identifier.
- MinHeightGlobal: Minimal height of tokens in the entire repository hierarchy.
- MinHeightSub[]: Vector determining the minimal token height for each child domain.
- ExpDate: This field defines when the group record expires (see Section 7 for details).

In contrast to TRS-RC, leaf repServers store a group record, too, consisting only of *Group*, *MinHeightGlobal* and *ExpDate*. *MinHeightGlobal* determines the minimal height of all tokens in the entire repository hierarchy. If the height of a locally available token is not greater than *MinHeightGlobal*, a local token can be delivered to the requesting client. Otherwise, a token search has to be performed to find a token with minimal height, i.e. a token with height equal to *MinHeightGlobal*.

A token search of TRS-MH consists of two phases, too. In the first phase of the token search, a *TokenRequest* is forwarded in leaf-to-root direction until a repServer is found that is part of the group tree and the following condition holds: min(*MinHeightSub*) ≤ *MinHeightGlobal*. If this condition holds at a repServer, a minimal-height token can be found in this repServer's subhierarchy, i.e. at one of the transitive children, and hence the search domain needs no further enlargement.

In the second search phase, each repServer forwards the *TokenRequest* message to a child *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 minimal height from *Group*'s token basket and delivers it directly to the searching leaf repServer, without using the TRS hierarchy.

If a repServer delivers a token to another repServer or a client, *MinHeightGlobal* of this repServer is updated to the token's height if it is greater than *MinHeightGlobal*. If this repServer is not already connected to the group tree or *MinHeightGlobal* or the minimal height of all locally available tokens has changed, it sends a *HeightUpdate (minimal local token height, MinHeightGlobal)* to its parent. A non-leaf repServer receiving a *HeightUpdate* message checks whether the corresponding group record already exists and creates one otherwise. Then the receiver updates *MinHeightSub*[i], where i was the sender of *HeightUpdate*, to the received value of the minimal token height and *MinHeightGlobal* to the received value of *MinHeightGlobal*, provided that *MinHeightGlobal* is not decreased. The forwarding of *HeightUpdate (Group, min(MinHeightSub), MinHeight-*

Global) to the parent in the TRS hierarchy is then repeated as long as at least one of min(*MinHeightSub*) or *MinHeightGlobal* undergoes a change and the root node is not already reached. The details of TRS-MH are described in [14].

7. FAULT TOLERANCE OF THE TOKEN REPOSITORY SERVICE

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

When a group management operation is to be performed and the home repServer is not available caused by a crash or network partition, any other repServer can be selected to execute those operations. To be able to select another repServer, each client maintains a list of some alternative repServers. Of course, when selecting another repServer, those that are in close domains are preferable. Note that using another repServer results in larger distances between parent and child nodes in the created ACK tree, which increases network load and delay for the reliable multicast protocol.

All token information is maintained according to the soft state principle. 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. The mechanism for group records used in TRS-RC and TRS-MH is analogous. In most cases the updating of group records through a *TokenAvail* or *HeightUpdate* message due to a change in the group tree is sufficient to extend the group record's expiration date of the parent node. If group tree changes are too infrequent, additional *TokenAvail* or *HeightUpdate* messages are sent to prevent the parent's group record from being discarded.

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.

The mechanisms described above ensure that tokens are not permanently lost. However, token loss can result in higher overhead for finding a token in the TRS and disadvantageous ACK trees. For example, assume that the home repServer of a node has crashed. Then a token from another domain is used in case the home repServer is not yet available or the tokens are not yet recovered. This leads to larger distances in the ACK tree between parent and child nodes, which increase network load and delay for the reliable multicast transport protocol.

As this refresh 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 additional communication overhead is low.

8. PERFORMANCE ANALYSIS

In this section we present some analytical results comparing the TRS strategies with ERS and ERA. We have evaluated the maximum message overhead and the maximum height of the created ACK tree.

8.1 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 n_j for joining a group is therefore as follows:

$$n_j = search\ messages + reply\ messages$$

$$= 255_m j + \sum_{i=1}^{j} i$$

$$= 255_m j + \frac{j^2 + j}{2}$$

Index *m* identifies multicast messages and *j* is the number of join operations (see Table 2). 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-PS is one message, the repCreateGroup message. To delete a group, *repDeleteGroup* is sent to the home repServer which sends one multicast *DeleteGroup* message to all other repServers.

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:

Fehler! Es ist nicht möglich, durch die Bearbeitung von Feldfunktionen Objekte zu erstellen.

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

$$n_j \approx 2j$$

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 message overhead for creating a group using TRS-RC is one message to the home repServer and (*t*-1) messages to establish the group tree, where *t* is the height of the TRS tree. To delete a group, *repDeleteGroup* is sent to the home repServer which forwards it along the edges of the group tree. The number of messages is therefore:

$$n_d = \sum_{i=0}^{t-1} k^i$$

To determine the number of messages for joining a group we distinguish between the first (*B*-1) joins and the remaining ones. We assume, that the first (B-1) joins are issued at different home repServers and therefore result in a token search. In worst case, the token search must be forwarded to the root node and from the root node to a leaf node which results in 2(t-1) messages. If the token is handed over, the group tree must be updated which results in (t-1) messages. 3 messages are necessary to send *repJoinGroup* to the home repServer, hand over a token to the searching repServer and finally return the token to the client. After the first (B-1) joins all remaining ones involve no further token search and therefore result in only 2 messages per join operation:

$$n_j = \sum_{i=1}^{t-1} [(k^i - k^{i-1})(3(t-i)+3)] \\ + (j - B + 1)2 \quad (j > B)$$

If a client leaves a group this can result in a necessary update of the group record. So, in worst case t messages are necessary.

Now we want to analyse the TRS-MH strategy. The number of messages to create a group, delete a group and leave a group are equal to TRS-RC. In worst case the *repJoinGroup* operation results everytime in a token search and update of the group records:

$$n_j = j(3(t-1)+3)$$

Figure 6 depicts the maximum number of sent messages for the ERS, TRS- PS, TRS-RC and TRS-MH approaches. The number of join operations range from 1000 to 10000. Note that the y-axis has a logarithmic scaling. In this scenario it is assumed, that one group is created, the depicted number of nodes on the x-axis join this group and half the number of joining nodes leave the group. The results show that ERS cannot be applied for large receiver groups. The TRS approaches provide significantly better scalability. The best scheme in terms of message overhead is TRS-RC. In contrast to ERS and TRS-PS it sends no multicast search messages and in contrast to TRS-MH it results in less overhead to search for a token in the repServer hierarchy and update the group records.

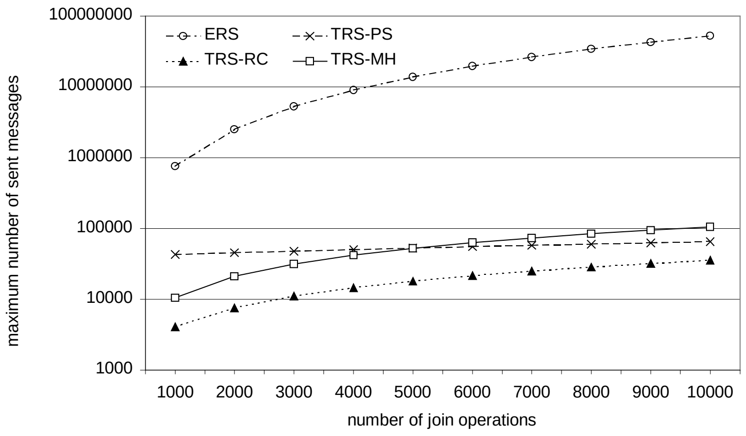


Figure 6: *Maximum number of sent messages*

8.2 Tree height

The height of the created ACK tree influences its reliability and round trip delay. Desirable are trees with low height (see Section 6). 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:

Table 2. *Notation and summary of analytical results*

n_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.
B………..	Number of (leaf) repServers requested for a token.
N………..	Number of nodes in a complete k-ary tree.
h…………	Height of the created ACK tree.
t…………	Height of the TRS hierarchy.

Maximum message overhead with ERS:

$$n_c = 0, \quad n_d = 0, \quad n_j = 255_m j + \frac{j^2 + j}{2}, \quad n_l = 0$$

Maximum message overhead with TRS-PS:

$$n_c = c, \quad n_d = d + d_m, \quad n_j = 2j + 255(B-1)_m + B^2 - 1 \approx 2j \quad (j >> B), \quad n_l = l$$

Maximum message overhead with TRS-RC:

$$n_c = ct, \quad n_d = \sum_{i=0}^{t-1} k^i, \quad n_j = \sum_{i=1}^{t-1} [(k^i - k^{i-1})(3(t-i)+3)] + (j-B+1)2 \quad (j > B), \quad n_l = lt$$

Maximum message overhead with TRS-MH:

$$n_c = ct, \quad n_d = \sum_{i=0}^{t-1} k^i, \quad n_j = \sum_{i=1}^{t-1} [(k^i - k^{i-1})(3(t-i)+3)] + (j-B+1)(2+t-1) \quad (j > B), \quad n_l = lt$$

Maximum tree height with ERS/ERA:

$$h = j - l + 1$$

Maximum tree height with TRS-PS and TRS-RC:

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

Maximum tree height with TRS-MH:

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

Number of nodes in a complete k-ary tree:

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

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

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

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

Tree height with j join operations:

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

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:

$$\begin{aligned} h &= B-1+\text{height of a complete k-ary tree} \\ &\quad \text{for } (j-(B-1)) \text{ join operations} \\ &= B-1+\lceil \log_k((j-l-B+2)(k-1)+1)\rceil \end{aligned}$$

The maximum tree height for TRS-RC is equal to the results of TRS-PS. For TRS-MH the tree height is equal to the height of a balanced tree:

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

Figure 7 depicts the maximum resulting ACK tree height of 1000 to 10000 join operations. The result of TRS-MH is equal to the height of a balanced *k*- bounded tree and therefore optimal. ERS and ERA can result in large tree heights, which is disadvantageous for round trip delays in the ACK tree and reliability.

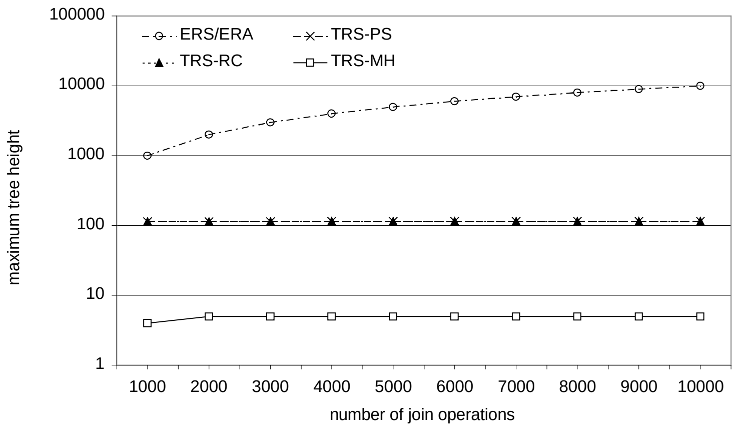


*Figure 7:	Maximum resulting ACK tree height*

9. SIMULATIONS

We have performed simulations using the NS2 network simulator [1] to compare the token repository service with expanding ring search strategies. The networks are generated with the network generator Tiers [3].

Fig. 8 depicts the dependency between received messages and various levels of background load. 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, i.e. a background load of 100% means that each network link was busy 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 strongly. This behaviour 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 sender of ERS starts a new multicast message with increased TTL. Note that 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 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 strongly 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 Figure 8 the message overhead of ERS with 5s timeout interval is up to 6% 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.

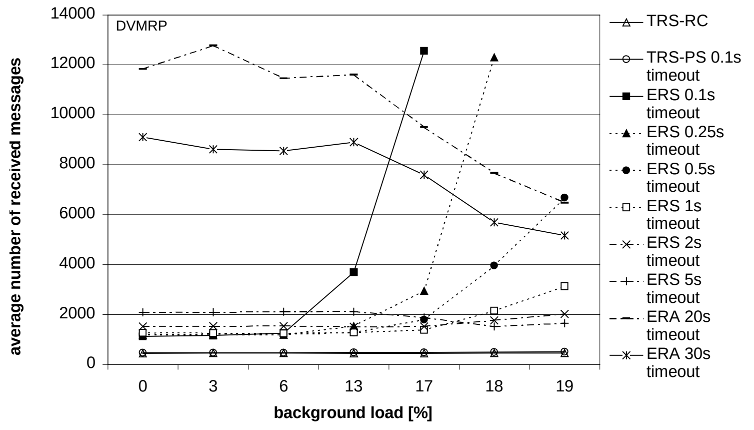
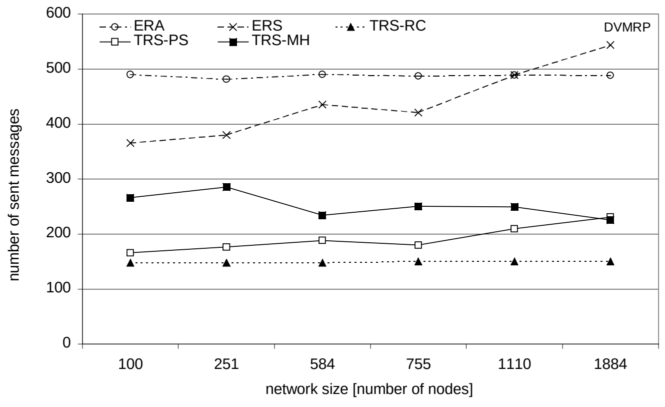


Figure 8: Messages received depending on the background load

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 therefore 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 token repository services, with random-choice strategy and proxy server strategy, is much lower compared to ERS and ERA and moreover, independent of the background load, always constant. The minimal-height strategy, which is not included in the figure, results in about the same message overhead as the random-choice strategy. We have also simulated the proxy server strategy with various timeout intervals but the results have differed only slightly.



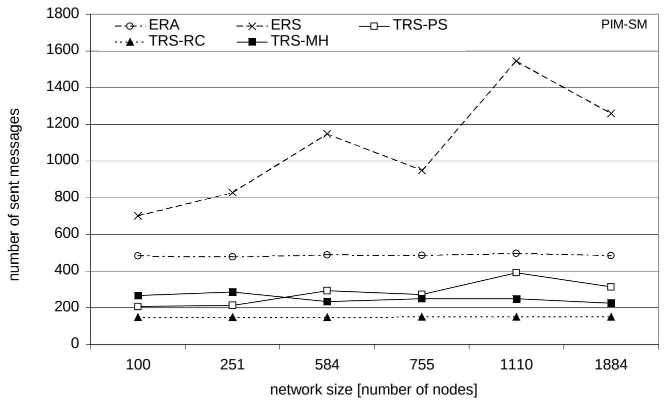
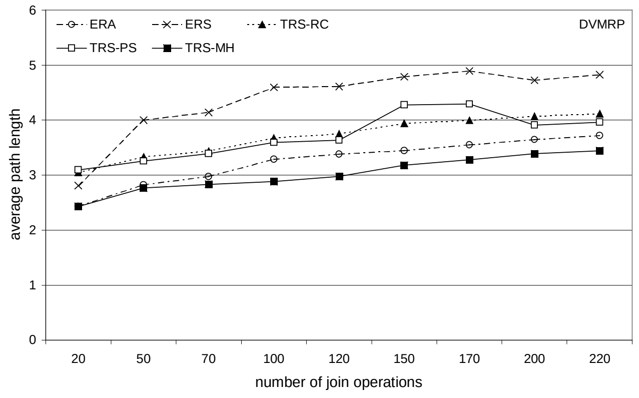


Figure 9: Messages sent depending on the network size with DVMRP and PIM-SM

Figure 9 shows the results of another simulation study to determine the influence of network size on scalability. 50 join operations are simulated with DVMRP and PIM-SM routing. The message overhead of TRS-RC, TRS-MH 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 TRS- MH 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.

The last simulation results depicted in Figure 10 investigate the average path length respectively height 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



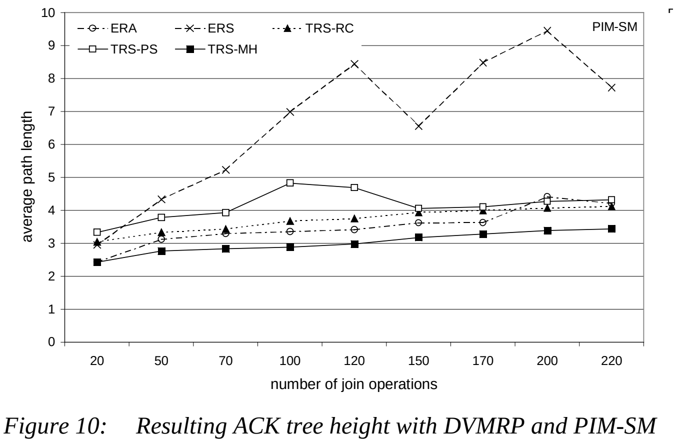


Figure 10: Resulting ACK tree height with DVMRP and PIM-SM

equivalent to the average number of nodes that must rejoin the tree if a single ACK tree node fails. Furthermore, low tree height results usually in low delays in the ACK tree.

Figure 10 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. Note that TRS-MH creates ACK trees with minimal path length. 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. This is caused by the characteristic of PIM- SM that multicast messages are always disseminated from the same core node in the network. Therefore, ERS finds always nodes close to this core and nodes that are far away from the core node get no child nodes, which results in large tree height.

10. SUMMARY

In this paper we have presented the token repository service, which is an efficient and robust approach for constructing ACK trees. The basic concept of our approach is a distributed repository storing tokens, which represent the right to connect to a certain node in an existing ACK tree.

Compared to the various approaches based on expanding ring search, the TRS has several advantages. It needs no bidirectional multicast support for joining nodes and produces network load only when a receiver joins a group. Furthermore, using the TRS, the undesired impact of the multicast routing protocol on the ACK tree construction in terms of scalability and quality of the created ACK trees is almost eliminated. In contrast to ERS with DVMRP, the TRS needs no separate routing tree to be established for each receiver joining the group. In contrast to ERS with PIM-SM, the disadvantageous core based dissemination of multicast messages hardly influences the results of the TRS. We can conclude from the presented simulation results that the TRS appreciably improves scalability. But also in terms of round trip delay and reliability of the created ACK trees, the proposed TRS performs in many cases better than ERS approaches.

REFERENCES

[1] Bajaj S., Breslau L., Estrin D., Fall K., Floyd S., Haldar P., Handley M., Helmy A., Heidemann J., Huang P., Kumar S., McCanne S., Rejaie R., Sharma P., Varadhan K., Xu Y., Yu H., Zappala D.: Improving simulation for network research, Technical Report 99- 702, University of Southern California, 1999.

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

[3] Calvert K., Doar M.B., Zegura E.W.: Modelling internet topology, IEEE Communications Magazine, June 1997.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

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

[11] Maihöfer C.: Improving multicast ACK tree construction with the Token Repository Service, IEEE ICDCS Workshop, 2000, pages C57-C64.

[12] Maihöfer C.: Scalable and reliable multicast ACK tree construction with the Token Repository Service, Proceedings of the International Conference on Networks (IEEE ICON), 2000, pages 351-358.

[13] Maihöfer C.: A bandwidth analysis of reliable multicast transport protocols, to appear in Proceedings of the Second International Workshop on Networked Group Communication (NGC), 2000.

[14] Maihöfer C., Rothermel K.: Constructing height-balanced multicast acknowledgment trees with the Token Repository Service, Technical Report 1999/15, University of Stuttgart, Faculty for Computer Science, 1999.

[15] Maihöfer C., Rothermel K., Mantei N.: A throughput analysis of reliable multicast transport protocols, to appear in Proceedings of the Ninth International Conference on Computer Communications and Networks (IEEE ICCCN), 2000.

[16] 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.

[17] 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 ICCCN'99), 1999, pages 139-145.

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

[19] Whetten B., Taskale G.: An overview of the reliable multicast transport protocol II, IEEE Network, 14(1), 2000, pages 37-47.

[20] 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.