Improving 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

Many new applications like groupware systems, news and file distribution or audio/video systems are based on multicast as a prerequisite for scalability. Many of these applications need the multicast support to be reliable, which is realized in a scalable way by tree based multicast transport protocols, where the receivers are organized in a 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 creating ACK trees which is based on the concept of a distributed token repository service. Simulation results show that our approach leads to a lower message overhead than ERS and results in better shaped ACK trees.

1. Introduction

Multicast support is a prerequisite to ensure scalability for large receiver groups. Although multicast support is already available in the Internet, the provided IP multicast service [1, 2] offers only best effort semantics. Several protocols have been proposed to overcome this drawback by a protocol layer on top of IP multicast [3, 4, 5, 6, 7, 8].

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. Some protocols use forward error correction (FEC) to add redundancy and hence improve the probability for correct message delivery. However, if delivery is to be guaranteed this must always be combined with the acknowledgment concept.

The class of sender- and receiver-based reliable multicast protocols [3, 4], where all receivers send their positive or negative acknowledgment (ACK) messages to the source, can cause the well-known acknowledgment implosion problem. In large multicast groups, with many hundreds or even thousands of members, the source may be overwhelmed by the amount of acknowledgments. The ACK implosion problem is a critical challenge for reliable multicast solutions.

To overcome the ACK implosion problem, the most promising approaches are tree-based protocols [5, 6, 7, 8]. They ensure scalability by organizing all group members in a so-called ACK tree. Although several variations of tree-based reliable multicast protocols are available, mainly distinguished by their use of positive or negative acknowledgments, all solutions share the same concept. Instead of sending an acknowledgment message directly to the sender, each receiver confirms the correct delivery only to its predecessor in the ACK tree, which is responsible for possible retransmits. An inner node in the ACK tree collects all acknowledgments. After it has received the multicast message and the corresponding ACK from each successor, it sends an aggregated ACK to its predecessor, 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 successors, no node and no part of the network is congested by 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 to discover a suitable predecessor 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 the 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 causes particular problems in combination with unidirectional core base 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 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 predecessor node in the ACK tree. A node joining a group asks the TRS for a token of this group, which identifies the predecessor 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 in Section 4 we present simulation results before we conclude with a brief summary in Section 5.

2. Problem definition and related work

Reliable multicast protocols can be classified into sender-based, receiver-based, ring-based and tree-based approaches. In a sender-based protocol, the sender is responsible for error control. The receivers send positive ACKs to the sender which allows to detect errors by missing ACKs. In large groups, this protocol class results in a large number of ACK messages and consequently in an overwhelmed sender.

Usually, the probability for delivering a packet correctly is higher than the probability for an error, hence the use of negative ACKs (NAKs) to indicate missing packets instead of positive ones for every correct packet results in a lower message overhead. The class of receiver-based protocols uses this approach where the error detection must be done by the receivers. If the retransmits are also sent by multicast, it is sufficient for the sender to detect that at least one receiver needs a retransmit. Therefore, a NAK avoidance scheme [4] can be used, in which a receiver that has detected an error sends a NAK via multicast provided that not already a NAK for the same data has been already sent by another receiver. Although receiver-based protocols scale better than sender-based ones, they have some other drawbacks. One drawback known as the crying-baby problem [4] is its dependency on the most unreliable receiver or most unreliable path to a receiver, respectively. If one receiver permanently detects message loss and multicasts NAKs, all of this group's members are congested. Another drawback is the non-determinism since a sender cannot decide whether a message has been received by all receivers and therefore can be discarded, or if a NAK will be still received later.

In ring-based protocols [9], all group members are organized in a virtual ring structure, where one member in the ring has the role of the retransmitter for lost messages. The scalability is improved by passing the retransmitter role from time to time to the next member in the ring. This mechanism is also useful to ensure a global ordering of events. Since each member must maintain a membership list and the ring management is complex, ring based protocols have a limited scalability.

Tree-based protocols offer the best scalability for large groups, since each group member is involved in the error control scheme with an acceptable small part. To overcome the ACK implosion problem all group members are organized in a ACK tree, where each inner node is responsible for its subhierarchy. Instead of sending an acknowledgment message directly to the sender, each receiver confirms the correct delivery only to its predecessor in the ACK tree. Each non-leaf node in the ACK tree collects all acknowledgments from its direct successor nodes. If an ACK is missing at a non-leaf node it sends a retransmit to the corresponding node. Depending on the used transport protocol an inner node sends an ACK to its predecessor after it has received the multicast message or sends an aggregated ACK after it has received the multicast message and the corresponding ACK from each successor, confirming the correct message delivery for the entire subhierarchy. If finally the root node has received all ACKs from its direct successors nodes, the reliable message delivery is ensured.

Each node in the ACK tree has an upper bound on the number of its successors, therefore no node and no part of the network is congested by messages. We will call a node *k-bounded* if it can accept at most *k* successor nodes.

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 predecessor that is not already *occupied*, i.e. has not already *k* successors. Most approaches to establish an ACK tree are based on expanding ring search (ERS). ERS is a common technique to search for resources in a network [10]. With the basic ERS approach for setting up ACK trees, the joining node looks for a predecessor in the ACK tree by sending multicast search messages with increasing search scopes [5]. The first message is sent with a time-to-live (TTL) of one, i.e. it is limited to the sender's LAN (see Fig. 1.). 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 predecessors 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 successor nodes with multicast invitation messages [6, 8]. The invitation messages can be sent with

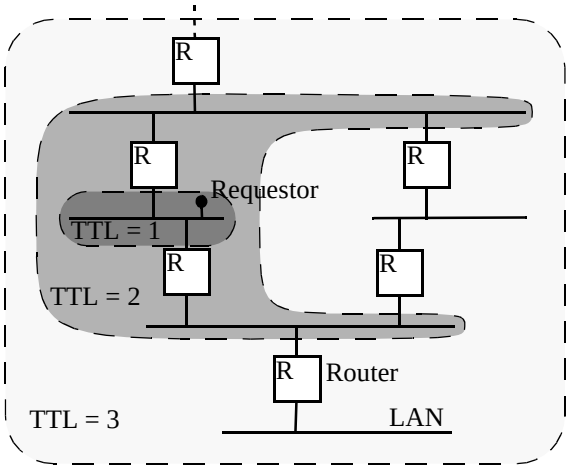

Fig. 1. Increasing search radius of ERS

fixed TTL [6], increasing TTL [11] or according to a special distribution function [8]. We will call this method expanding ring advertisement (ERA) [8]. Some protocols use a combination of both approaches [7, 11].

ERS and ERA approaches have the great advantage of fault tolerance. On the other hand, our simulation results in Section 4.2 will show that both result in a huge 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) [12] 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 [13], the use of ERS and ERA result in a traffic concentration at the core, an even higher message overhead and ACK trees of poor quality. 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.

In the following sections we will propose the token repository service with proxy server strategy (TRS-PS), which is based on a combination of a repository service and the well-known ERS. In [14] we have already described 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

In this section, we will describe the interface and concept of the token repository service, which is our proposed infrastructure for building up ACK trees. The details of the implementation will be covered in the next section.

A token stored in the TRS represents the right to connect to a particular 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 predecessor 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 TRS-PS strategy

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 2 is responsible for domain 1 consisting of nodes N_{1x}.

Note that nodes can be arbitrarily assigned to domains and that the repServer does not have to be located inside its domain. However, 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

Table 1. Operations provided by the TRS

Operation	Description
repCreateGroup (Group, Ack-Root, K)	This operation makes *Group* known to the repository service. *AckRoot* identifies 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*-bound. The operation returns a token identifying the predecessor 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* into the repository. It is called when a successor of *Owner* disconnects from the *Group*'s ACK tree.

domain should be typically smaller than between two nodes in different domains [15]. Furthermore, the repServer should be located inside its domain. A repServer, responsible for all nodes in its domain, is called these nodes' *home repServ*er. In Figure 2, S_1 is the home repServer of nodes N_{1x}. Nodes access the token repository service only via their home repServer.

Tokens are stored on 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 successor 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. Consequently in general 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.

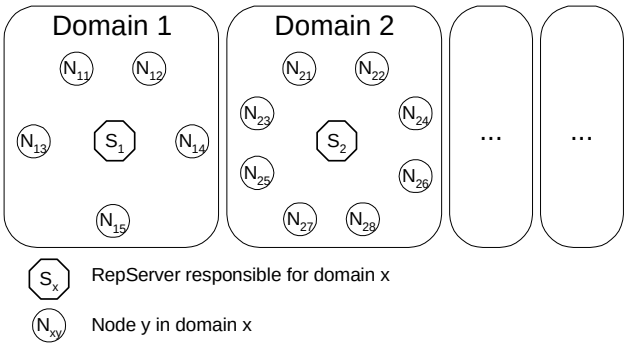

Fig. 2. Domain structure

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*: This field identifies the corresponding multicast group in a unique way.
- *SetOfTokenPackets*: The tokens in the token basket are grouped according to their owner into so-called token packets.

Each token packet includes the following information:

- *Owner*: This field identifies the owner of the token packet uniquely. A node receiving a token from this packet is allowed to connect to *Owner* in the corresponding ACK tree.
- *Height*: This field specifies the height of the *Owner* in the corresponding ACK tree. This information can be 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*: This field specifies the amount of tokens in this token packet.
- *ExpDate*: This field defines when the token packet expires.

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.

The token information is maintained according to the soft state principle [16]. Token packets are associated with an expiration date. If an expiration date is reached, it must either be extended or the token packet will be discarded automatically. Although our protocols discard token packets explicitly during normal operations, this mechanism allows to design a robust protocol, ensuring even in the presence of node and communication failures that eventually all outdated information is removed. In addition, having such a mechanism in place allows to use light-weight protocols for explicitly deleting token packets.

If a token packet expires it must be checked, whether it is still valid and thus the expiration date should be extended, or whether the associated token packet has to be removed. Obviously, the lifetime of a token packet depends on the lifetime of its owner. When a token packet expires, the repServer storing this packet asks the token packet's owner to extend the expiration date. If the owner responds, the expiration date is extended accordingly, otherwise the entire packet is removed. If this was the last token packet in this group's token basket, the token basket is deleted, too.

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.

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 [5, 6, 7, 8].

A node *N* creates a new multicast group by initiating a *repCreateGroup (Group, AckRoot, 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 2 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 3 depicts this scenario after the token basket is created.

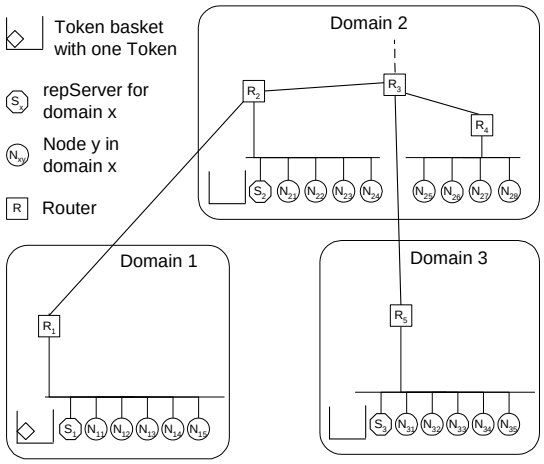

Fig. 3. Example scenario

When the operation *repDeleteGroup (Group)* is triggered 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 ensures that all state information is removed within the expiration time 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 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* 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 a 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 multicast messages with a TTL less than 5 do not reach other repServers or only S_2 which 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 searchs.

Note that if a repServer *R* replies with *TokenAvail* to a searching repServer *S*, the token is not reserved at *R*. This means that if *R* receives a *GetToken* message possibly no token for the requested group exists. In this case *R* replies with a *NoToken* message and *S* has to search for another repServer.

The internal operations for the token search procedure are summarized in Table 2.

Table 2. Internal TRS-PS operations

Operation	Description
TokenSearch (Group, Requestor)	A *TokenSearch* is 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)	*GetToken* is 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)	*NoToken* is 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. As we assume that a node has no descendants in the ACK tree when it leaves the group[1], 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 predecessor in the ACK tree. Conceptually if *N* leaves a group it releases its predecessor's token allocated by *N* so far. Hence *N*'s predecessor 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. Simulations

We have performed simulations to compare the token repository service with expanding ring search strategies. Before the simulation results are presented, we will describe the simulated scenario in the following subsection.

[1]. The used multicast transport protocol must ensure that a node is only allowed to leave a group if it has no successor nodes in the ACK tree, i.e. is a leaf node. An inner node can leave a group after it has arranged a rejoining for all successor nodes at other ACK tree nodes.

4.1. Simulation scenario

Our simulations are performed using the NS2 [17] network simulator. The networks were generated with Tiers [18]. All links are symmetrical duplex links with the bandwidths 10Mbps for LANs, 100Mbps for MANs and 1000Mbps for 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. 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.

TRS 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 [6]. In all simulations the maximum branching factor in the ACK tree is $k = 10$.

4.2. Simulation results

Our first simulation result in Fig. 4 depicts the dependency between received messages 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, i.e. a background load of 100% means that each network link was busy during the entire simulation.

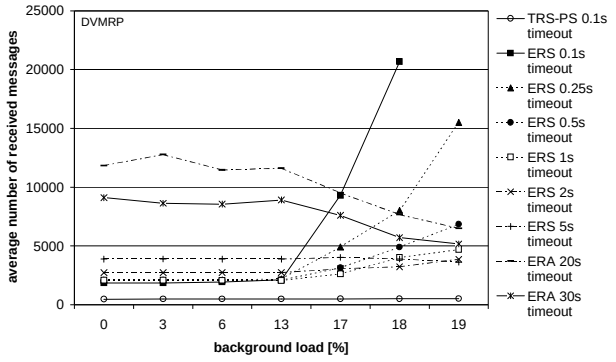

Fig. 4. Received messages

The results show that ERS scales poorly with the background load. If the background load exceeds a certain level, the number of received messages rises exponentially. This behavior is caused by increased message delays due to high background load. When the delay of a search and the resulting answer message exceeds the timeout interval, the sender of ERS starts a new multicast search 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 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 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 successor 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 that runs for a restricted simulated time period. The use of ERA leads to network congestion and consequently to a high message delay, therefore not all invitation messages were delivered during the simulated time.

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

Fig. 5 illustrates the average path length of the created ACK trees. The path length affects the delay and reliability of the created ACK tree. The multicast service may be disrupted for a node if one of its predecessors in the ACK tree becomes unavailable. Therefore, the lower the number of predecessors the higher the reliability from this node's perspective. So, the average path length of the ACK tree can be used as a quality 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. 5 shows that TRS-PS as well as ERA lead to ACK trees with low path lengths that are near to the theoretical

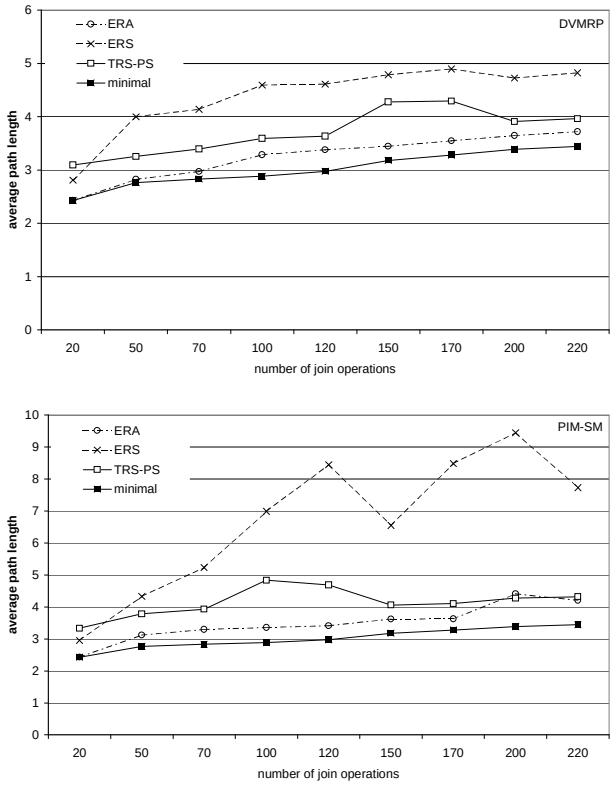

Fig. 5. Path length of the created ACK tree

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 successors.

5. Conclusions

In this paper we have presented the token repository service with proxy server strategy (TRS-PS), which is an efficient and robust approach for constructing ACK trees. The basic concept of our approach is a distributed token repository combined with the fault tolerant ERS approach. The repository service stores 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, TRS-PS has several advantages. It needs no bidirectional multicast support for joining nodes and produces network load only when a receiver joins or leaves a group. Furthermore, using TRS-PS, 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. We can conclude from the presented simulation results that TRS-PS appreciably improves scalability and improves in many cases the quality of the created ACK trees.

References

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

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

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

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

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

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

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

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

[9] Whetten, B.; Kaplan, S., Montgomery, T.: A high performance totally ordered multicast protocol, available from http://147.46.59.102/~imhyo/papers/multicast/RMP_dagstuhl.ps.

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

[11] 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.

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

[13] Estrin, D.; Farinacci, D.; Helmy, A.; Thaler, D.; Deering, S.; Handley, M.; Jacobson, V.; Liu, C.; Sharma, P.; Wei, L.: Protocol independent multicast-sparse mode (PIM-SM): protocol specification, RFC 2362, 1998.

[14] 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), 1999.

[15] Theilmann, W.; Rothermel, K.: Dynamic distance maps of the Internet, Proceedings of the Conference on Computer Communications (IEEE Infocom), 2000.

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

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

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