A Benchmarking Environment for Performance Evaluation of Tree-based Rekeying Algorithms

Abstract

While a vast number of solutions to multicast group rekeying were published in the last years, a common base to evaluate these solutions and compare them with each other is still missing. This paper presents a unified way to evaluate the performance of different rekeying algorithms running on the server side. A rekeying benchmark estimates rekeying costs from a system point of view, which allows a reliable comparison between different rekeying algorithms. For this purpose, new system metrics related to rekeying performance are defined: the Join Rekeying Time (JRT) and the Disjoin Rekeying Time (DRT). By means of four simulation modes, these metrics are estimated in relation to both the group size and the group dynamics. A benchmark prototype, implemented in Java, demonstrates the merit of this unified assessment method by means of two comprehensive case studies.
Keywords: Group Rekeying Algorithms, Performance Evaluation, Benchmarking, Simulation

1. Introduction

While multicast is an efficient solution for group communication over the Internet, it raises a key management problem when data encryption is desired. This problem originates from the fact that the group key used to

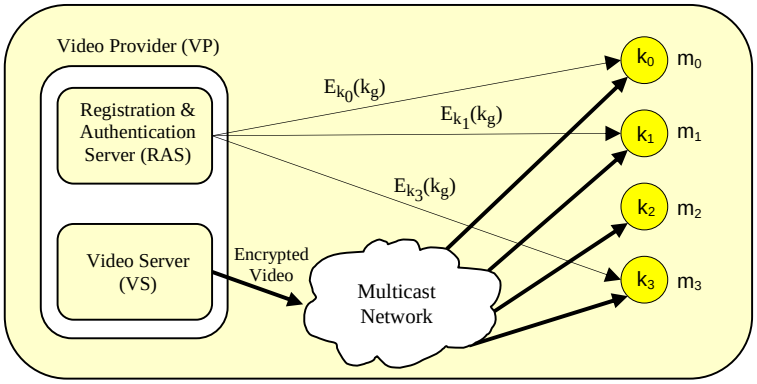


Figure 1: Pay-TV a potential scenario for secure multicast

encrypt data is shared between many members, which demands the update of this key every time a member leaves the group or a new one joins it. The process of updating and distribution of the group key, denoted as group rekeying, ensures forward access control regarding leaving members and backward access control concerning the joining ones. Figure 1 represents a Pay-TV environment as an example for a multicast scenario. A video provider (VP) utilizes a video server (VS) to deliver video content encrypted with a group key k_g. A registration and authentication server (RAS) manages the group and performs group rekeying. Every registered member gets an identity key k_d (e.g., k_0 to k_3 in Figure 1) and the group key k_g. To disjoin member m_2, for instance, the RAS generates a new group key and encrypts it with each of the identity keys of the remaining members. In other words, disjoining a member from a group having n participants costs a total of $n-1$ encryptions on the server side. Obviously, a scalability problem arises.

This problem has been amply addressed in the last decade to reduce the rekeying costs and to achieve a scalable group key management. A large

variety of architectures, protocols, and algorithms have been proposed in literature, see [1] to [14]. Although many approaches have been proposed, the reader of related publications lacks a way to compare the results of these solutions to each other. This is attributed mainly to vastly different ways of estimating rekeying costs by different researchers and to the application of highly diverse metrics to express these costs. In the scope of this work, this problem is denoted as the *rekeying performance evaluation problem*.

In this paper a *rekeying benchmark* is presented, which allows for a *reliable rekeying performance evaluation* and for fair comparison between different rekeying algorithms. This originates from evaluating the rekeying performance on a high abstraction level: Rekeying costs are determined on the system level independent of the evaluated rekeying algorithms themselves and regardless of the underlying cryptographic primitives and execution platform. The rekeying benchmark is realized by a simulator, which supports the execution of different rekeying algorithms with uniform simulation parameters. The simulator estimates unified cost metrics and presents simulation results in the same diagram for comparison.

Group key management schemes may be centralized or distributed. Centralized schemes rely on a signle trusted point and are, therefore, more appropriate for applications with high security requirements such as banking or paid services such as Pay-TV over IPTV. The proposed benchmark deals with rekeying algorithms for this type of key management schemes.

The rekeying benchmark considers only the costs of cryptographic operations required for rekeying on the server side, which dominate the total cost in most cases. Including other factors is part of future work.

The rest of the paper is organized as follows. Section 2 details the rekeying performance evaluation problem. Section 3 represents the design concept of the proposed rekeying benchmark and Section 4 describes the realization of this concept as a simulation environment. Section 5 details the benchmark design and Section 6 outlines its implementation. To show the advantage of this solution Section 7 provides two case studies.

2. Rekeying Performance Evaluation Problem

A typical problem in scientific work relates to the analysis and evaluation of own results and comparing them with those of related work. This problem, on the one hand, can be attributed to the increasing number of scientific institutions and the vast publication possibilities. This trend hinders a comprehensive overview of the state-of-the-art situation in some scientific field. On the other hand, some research areas - because of their novelty, complexity, or both - lack a unified way of drawing these comparisons. This situation, for instance, does not apply to Advanced Encryption Standard (AES) [16] due to the availability of recognized performance metrics for block ciphers which are throughput and latency. On the contrary, such unified metrics are still missing for estimating the performance of rekeying algorithms, which is caused by both the novelty of this problem area and its complexity. This complexity, however, did not only result in largely different metrics to express rekeying performance, but also in diverse ways of estimating these metrics. In this respect, the reader of proposed work on multicast group rekeying is not only confronted with different performance quantities, but also with various estimation methods such as analytical modeling, simulation-based

approaches, and real-time measurement using provisional prototypes. Each one of these techniques has specific constraints and drawbacks, which can be outlined in the following points:

Analytical approaches. These approaches are always based on simplified models and relate to special cases such as full balanced trees in tree-based approaches. As a rule, rekeying performance can only be expressed by abstract numbers of some primitive operations, e.g., the number of encryptions for borderline cases, such as worst-case or best-case analysis.

Simulation-based approaches. These approaches are mostly used to prove a presented analytical investigation without model enhancement and without including sophisticated effects such as group dynamics.

Measurement approaches. These approaches deliver results, which are strongly dependent on the deployed cryptographic primitives, their implementation, and on the platform they run on.

Furthermore, the performance of group rekeying is influenced by several factors reflecting the group state and dynamics, on the one hand, and by some algorithm-specific parameters such as the tree degree in tree-based solutions, on the other. Accordingly, two questions arise for performance estimation: Which factors must be taken into consideration, and how should they be included as variables or as parameters? Again, largely different answers are given to these questions in related work.

In summary, the difficulty of evaluating different rekeying algorithms is attributed to the following three points:

1. Non-unified performance estimation methods.

2. Non-unified consideration of the input quantities affecting the performance.
3. Non-unified definition of output metrics representing the performance.

Table 1 gives a representative view of this situation in related work. Note that an input quantity can be considered either as a variable or as a parameter. A parameter is a variable which is kept constant in some evaluation process.

The different ways of looking at rekeying performance do not only obstruct an objective assessment of the corresponding algorithms, but also give an explanation of some inconsistencies in the conclusions drawn by some related work. Section 7 details this situation by means of two case studies.

3. Rekeying Benchmark Design Concept

3.1. Benchmark Abstraction Model

Rekeying is a solution for group key management in secure multicast. As an essential step in the process of joining and removing members, rekeying performance directly influences the efficiency of this process with major effects on the system behavior. The faster a member can be removed the higher is the system security. The faster a member can be joined the higher is the system quality of service. The performance of a rekeying algorithm directly affects the supportable group size and dynamics. Accordingly, the importance of rekeying performance results from its significance for the system behavior with respect to the following characteristics:

1. Amount of *quality of service* that can be offered to a joining member.

Work	Performance estimation method	Performance estimation mode and constraints	Input Quantities		Performance metric
			Variable	Parameter	
[3]	Analytical	Worst Case Average Costs	#Joins,#Disjoins,Tree degree, Group size		#Encryptions
	Simulation	Worst Case Average Costs	#Joins, #Disjoins, 0-1000 0-4000	Tree degree 2,4,8,16,32 Group size 1024,4096	
[8]	Measurement	Group grows from 5000 to 20000 with 1% probability for join and 0.1% for disjoin	Time 0-600 Sec 0-8000 Sec		#Messages per Minute, Tree levels
[7]	Analytical	Full balanced binary trees	Group size, Parameters for encryption, key generation and hashing costs		Abstract cost per join/disjoin, per multiple joins/disjoins
[1]	Analytical	Full balanced trees	Tree height and degree		#Encryptions per request
	Measurement	Join rate = Disjoin rate = 50%	Group size 0-8192	Tree degree 2-16	Request processing time
[9]	Simulation	Join rate = Disjoin rate = 50% n_0=10000 Worst Case Average Costs	Operation number 0-10000		Rekeying message cost per 2000 operations
[5]	Simulation	Statistically generated join/ disjoin patterns	Time 0-70 Min. 0-700 Min.	Batch period 0-40 Min. 0-50 Min.	Tree height
[2]	Analytical	Worst case best case analysis	Group size, tree degree, highest layer		#Keys per request
	Simulation	Full balanced trees	Group size 0-8192	#Cumulative layers	
[4]	Analytical		Group size,#merging members, #leaving members		#Exponentiations #Signatures #Verifications Time per join/disjoin Time per merging/partition
	Measurement	Average communication and client delay included	Group size 0-50	RSA module 512 bit, 1024 Bit	
[6]	Analytical	1 join / 1 disjoin	Group size All potential members Potential members not in the group currently		#Encrypted messages
	Simulation	Dedicated for some MBone sessions	Time 0-400 hours	Group sizes 4096, 64K Batch period 20-240 Min.	
[10]	Measurement		Payload size in bytes		Time

Table 1: Dissimilarity in rekeying performance estimation in related work

2. Amount of *security* against a removed member.
3. *Scalability* in terms of supportable group sizes.
4. *Group dynamics* in terms of maximal supportable join and disjoin rates.

These characteristics allow for evaluating rekeying performance on a high abstraction level, see Figure 2. To enable a reliable performance evaluation of rekeying algorithms, metrics should be used, which are independent of these algorithms. Therefore, the performance evaluation is settled on the *Benchmark Layer*, which is sperated from *Rekeying Layer* in Figure 2. The introduction of the *Cryptography Layer* and the *Platform Layer* is justified as follows. The Rekeying Layer performs join and disjoin requests based on cryptographic operations such as encryption and digital signing. For each cryptographic primitive a vast selection is available. Taking symmetric-key encryption as an example, rekeying may employ 3DES, AES, IDEA or other algorithms. The same rekeying algorithm behaves differently according to the utilized cryptographic primitives. Further more, the same cryptographic primitive features different performance according to the platform it runs on. This fact remains, even if public-domain libraries such as CryptoLib [17] are utilized to realize cryptographic functions. Consequently, a reliable rekeying benchmark does not only rely on an abstraction from the details of the analyzed rekeying algorithms. Rekeying itself must be decoupled from the underlying cryptographic primitives and from the executing platform.

The abstraction model of Figure 2 introduces essential design aspects for the benchmark:

1. The separation of rekeying algorithms from the cryptographic layer and from the execution platform leads to a substantial acceleration of

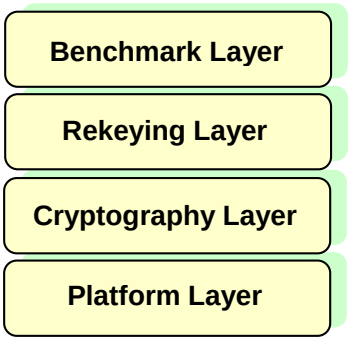


Figure 2: Rekeying benchmark abstraction model

the evaluation process. This gain is based on the fact that rekeying algorithms to be evaluated do not need to execute cryptographic algorithms anymore. Instead, they just provide information on the required numbers of these operations. The actual rekeying costs are then determined by the benchmark with the aid of timing parameters relating to the used primitives and the execution platform. This point will be detailed in the next section.

2. From the last point it is obvious that the demand for a reliable rekeying benchmark can not be fulfilled by real-time measurements on prototypes or final products, since these measurements can not be performed independently of the cryptographic primitives and the platform. Instead, for rekeying algorithms to be evaluated fairly and efficiently, some kind of simulation has to be employed.

3.2. Benchmark Data Flow

A good understanding of the benchmark abstraction model can be achieved by investigating the data exchange between its different layers as depicted in

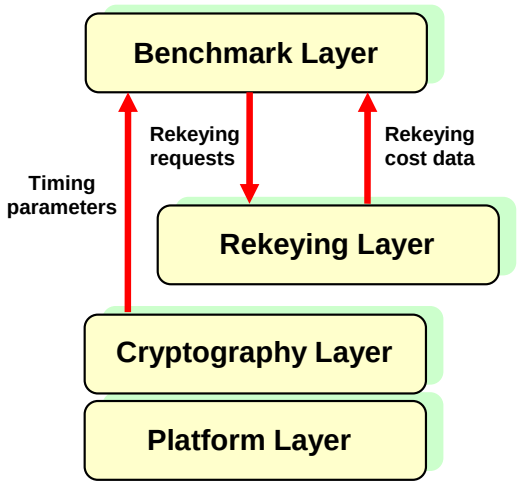


Figure 3: Data exchange in the rekeying benchmark

Figure 3:

1. The rekeying layer receives rekeying requests and executes pseudo rekeying, which means that rekeying algorithms only decide on the cryptographic operations needed for these requests without executing them. This issue is illustrated by the gap between the rekeying and the cryptography layers.
2. The rekeying requests are delivered without any timing information. This means that the rekeying layer is not informed about the temporal distribution of the rekeying requests. This task is assigned to the benchmark layer.
3. The rekeying cost data provide information on the number of the needed cryptographic operations for each rekeying request or request batch.

4. The timing parameters hide the cryptographic primitives and the executing platform to provide a unified cost estimation, which can be used by the benchmark layer for all rekeying algorithms in the same way.
5. To determine the time needed for executing a rekeying request, the benchmark sums up the products of the rekeying cost data and the corresponding timing parameters.

4. Rekeying Benchmark as a Simulation Environment

4.1. Cost Metrics and Group Parameters

Definition 1. ***Join Rekeying Time*** JRT

Join rekeying time is defined as the time elapsed between the arrival of a join request at the rekeying server and determining the group key by the new member. JRT can be determined as:

$$JRT = t^j_w + t^j_{comp,server} + t^j_{comm} + t^j_{comp,member} \tag{1}$$

where, t^j_w refers to the waiting time of the join request in the system queue, $t^j_{com,server}$ to the time elapsed by the rekeying algorithm to compute the rekeying message on the server, t^j_{comm} to the communication time required to deliver the rekeying message to the member, and $t^j_{comp,member}$ to the time elapsed by the member to compute the group key from the rekeying message.

Note that the time needed to propagate the rekeying request from a member to the rekeying server is not considered. This is because the delivery time of a rekeying request can be assumed as constant and thus has the same effect whatever rekeying algorithm is used.

Definition 2. ***Disjoin Rekeying Time*** DRT

Disjoin rekeying time is defined as the time elapsed between the arrival of a disjoin request at the rekeying server and determining the new group key by remaining group members. DRT can be determined as:

$$DRT = t^d_w + t^d_{comp,server} + t^d_{comm} + t^d_{comp,member} \tag{2}$$

While t_w^d, $t_{com,server}^d$ and t_{comm}^d are the counterparts of the join case, $t_{comp,member}^d$ is slightly different. Disjoin rekeying is completed when all remaining members get the new group key, so that data can be encrypted with this key and the access of the leaving member is denied. Therefore, we take $t_{comp,member}^d$ as the time needed to compute the group key by the worst-case remaining member. A Worst-case remaining member is the member located at the deepest leaf in the rekeying tree.

Definition 3. ***Rekeying Performance Metrics***

A rekeying performance metric is a collective term which refers to both JRT and DRT.

Definition 4. ***Group Parameters***

A group parameter is a collective term which refers to one of the following parameters:

1. *Group size* n : The number of group members.
2. *Join rate* λ : The arrival rate of join requests at the rekeying server.
3. *Disjoin rate* μ : The arrival rate of disjoin requests at the rekeying server.

4.2. Simulation Modes

The performance of rekeying algorithms in terms of join and disjoin rekeying times, JRT and DRT, differs depending on the group parameters n, λ, and μ. An algorithm with fast rekeying at high join rates, for instance, could work slower at high disjoin rates. Therefore, a detailed and fair evaluation of rekeying algorithms should be based on an appropriate selection of the group

parameters and their value ranges. For this purpose, the rekeying benchmark supports the following four simulation modes. See Table 2, which summarizes the main features of these simulation modes.

Transient Simulation. This simulation mode enables the evaluation of several rekeying algorithms for same group parameters. The performance metrics JRT and DRT are determined as a function of time in this case, i.e. $JRT(t)$ and $DRT(t)$. By this means, the behavior of rekeying algorithms over long time periods can be observed. To perform a transient simulation, an initial group size n_0, a join rate λ, a disjoin rate μ, and the desired simulation time t_{sim} are set by the user, see Table 2. Depending on λ and μ, the built-in request generator generates a list of requests over the simulation time t_{sim}. Each request is then sent to the rekeying algorithm to determine the rekeying message. Upon execution, the corresponding metric $JRT(t)$ or $DRT(t)$ is determined. This is repeated for all algorithms to be evaluated. The transient simulation is the foundation for all other simulation modes.

Scalability Simulation. The importance of this simulation mode results from the significance of the scalability problem in group rekeying. The scalability simulation investigates the effect of the group size on the performance of a rekeying algorithm. The user defines a group size range $[n_{start}, n_{end}]$, a simulation step Δn, and an observation interval T_o. For each value n of the range $[n_{start}, n_{end}]$, a transient simulation is performed with $n_0 = n$ and $t_{sim} = T_o$. The transient simulation thus results in a set of performance metrics $JRT(t)$ and $DRT(t)$ for each n. From all these metrics the worst-case values are considered, i.e., $JRT(n) = JRT(t)_{max}$ and $DRT(n) = DRT(t)_{max}$.

Join Dynamics Simulation. High join rates result in shorter inter-arrival times of join requests and higher computation overhead on the server. This causes longer waiting times for both join and disjoin requests, see (1) and (11). Thus, higher join rates affect not only JRT, but also DRT. The join dynamics simulation represents a way to investigate theses dependencies. The user defines an initial group size n_0, a disjoin rate μ, a fixed observation interval T_o, and join rate range $[\lambda_{start}, \lambda_{end}]$. For each value λ of this range, a transient simulation over T_o is started. As in the scalability simulation, for each λ several values for the performance metrics result, from which the worst-case values are selected, i.e., $JRT(\lambda) = JRT(t)_{max}$ and $DRT(\lambda) = DRT(t)_{max}$.

Disjoin Dynamics Simulation. The disjoin dynamics simulation is similar to the join dynamics simulation and can be exploited to investigate the behavior of rekeying algorithms under different conditions for the disjoin rate μ.

	Transient	**Scalability**	**Join Dynamics**	**Disjoin Dynamics**
Group parameters	n_0, λ, μ	λ, μ	n_0, μ	n_0, λ
Simulation parameters	t_{sim}	$T_0, \Delta n,$ $[\ n_{start}, n_{end}\ ]$	$T_0, \Delta\lambda,$ $[\ \lambda_{start}, \lambda_{end}\ ]$	$T_0, \Delta\mu,$ $[\ \mu_{start}, \mu_{end}\ ]$
Variable	time	n	λ	μ
Performance metric	$JRT(t)$ $DRT(t)$	$JRT(n)$ $DRT(n)$	$JRT(\lambda)$ $DRT(\mu)$	$JRT(\mu)$ $DRT(\lambda)$

Table 2: Simulation modes

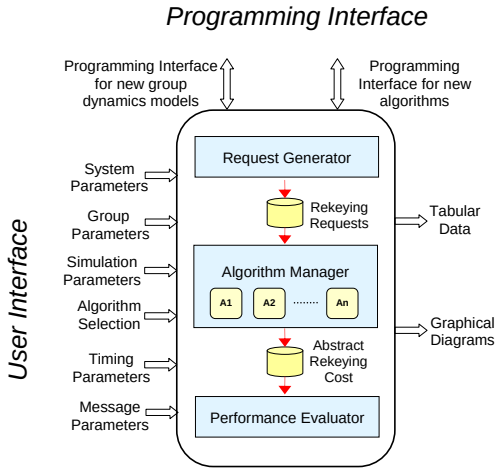


Figure 4: Benchmark architecture

5. Rekeying Benchmark Design

The rekeying benchmark is mainly composed of two interfaces and three components, as depicted in Figure 4. The *user interface* (UI) enables users to evaluate different rekeying algorithms by selecting these algorithms and setting the desired parameters. For designers a *programming interface* (PI) is provided to integrate new algorithms. In addition, groups with special dynamic behavior, which does not follow Poisson distribution, can be supported by means of a special programming interface module.

The component *request generator* creates a rekeying request list depending on the selected group and simulation parameters. An entry of this list keeps information on the request type, join or disjoin, the identity of the member to be joined or removed, and the arrival time of this request. The *algorithm manager* then selects and configures the rekeying algorithms according to the user settings. It coordinates all the functions of the simula-

tor and controls the rekeying algorithms. Based on the rekeying cost data delivered from the rekeying algorithms and the entered timing and message parameters, the *performance evaluator* finally determines the rekeying performance metrics JRT and DRT for each algorithm and prepares them graphical presentation.

5.1. Request Generator

The request generator produces a rekeying request list $RRL(T)$ by executing a main process (Section 5.1.1) and 3 subprocesses (Sections 5.1.2 and 5.1.3). Before describing these processes, we introduce some new concepts.

Definition 5. ***Rekeying Request***

A rekeying request is a 3-tuple (*type*, ID, t_a). *type* indicates the request type, which may be either join (J) or disjoin (D). ID represents the member identity to be joined (ID_J) or removed (ID_D). t_a describes the arrival time of a join (disjoin) request t_{aJ} (t_{aD}), measured from the simulation start point.

Definition 6. ***Rekeying Request List*** $RRL(T)$

A rekeying request list over T, $RRL(T)$, is an ordered set of rekeying requests, which arrive during a defined time interval T. The requests in the list are ordered according to their arrival times.

***Example* 1.** An $RRL(T)$ can be represented in tabular form. Table 3 depicts an example.

Definition 7. ***Join Arrival List*** $A_J(T)$

A join arrival list over a time interval T is an ordered list of inter-arrival times relating to all join requests generated during T: $A_J(T) = (\Delta t_J(1), \Delta t_J(2), \ldots,$

Request Type	Member Identity	Arrival Time (ms)
J	1099	0
D	50	0.1
D	178	2
J	22657	5.3

Table 3: Example for a rekeying request list $RRL(T)$

$\Delta t_J(h))$, where $\Delta t_J(i)$ indicates the inter-arrival time of the i-th join request in the interval T, and

$$\sum_{i=0}^{i=h} \Delta t_J(i) \leqslant T. \tag{3}$$

Definition 8. ***Disjoin Arrival List*** $A_D(T)$

Similarly to $A_J(T)$, a disjoin arrival list over a time interval T is defined as: $A_D(T) = (\Delta t_D(1), \Delta t_D(2), \ldots, \Delta t_D(k))$, where

$$\sum_{i=0}^{i=k} \Delta t_D(i) \leqslant T. \tag{4}$$

Definition 9. ***Member Identity*** ID

A member identity is a natural number between 0 and $n_{max} - 1$, where n_{max} refers to the maximum group size. This parameter is set during simulation setup. n_{max} is needed by some rekeying algorithms for setting up an appropriate tree data structure.

Definition 10. ***Complete Multicast Group*** M

A complete multicast group is the set of all the member identities: $M = ID(i)$, where $i \in 0 \ldots (n_{max} - 1)$.

Definition 11. ***Joined Multicast Subgroup*** M_J

A joined multicast subgroup is the subset of all the given identities. At

the start of a simulation with an initial group size n_0, M_J can be given as: $M_J = ID(i)$, where $i \in 0 \ldots (n_0 - 1)$.

Definition 12. ***Potential Multicast Subgroup*** M_D

A potential multicast subgroup is the subset of all the identities, which can be given to new members. At the start of a simulation with an initial group size n_0, M_D can be given as: $M_D = ID(i)$, where $i \in n_0 \ldots (n_{max} - 1)$.

5.1.1. Request Generator Process (GenReqList)

This process generates the rekeying request list $RRL(T)$ as given in Algorithm 1. First, the arrival process GetArrivalLists(T) is called to produce join and disjoin arrival lists $A_J(T)$ and $A_D(T)$. According to the inter-arrival times in these lists, the arrival times for the individual requests are determined. Depending on the request type, the member identity is then obtained by calling GetJoinID or GetDisjoinID. Then, the $RRL(T)$ is updated by the new request. After processing all entries in $A_J(T)$ and $A_D(T)$, the $RRL(T)$ is sorted with increasing arrival time. Note that the request generator is transparent to the simulation mode. Utilizing the generator for different simulation modes will be described in the scope of the Algorithm Manager. Example 2 illustrates Algorithm 1 in more details.

***Example* 2.** Assume a group of maximal 8 members, where 5 members are currently joined as follows: $M = \{0, 1, 2, 3, 4, 5, 6, 7\}$, $M_J = \{0, 1, 2, 3, 4\}$, $M_D = \{5, 6, 7\}$, See definitions 10, 11, and 12 for M, M_J and M_D, respectively. Assume that calling the process GetArrivalLists(T) on some interval T results in the inter-arrival time lists $A_J(T) = (10, 25)$ and, $A_D(T) = (11, 5, 7)$.

Algorithm 1 GenReqList

Require: T

Ensure: $RRL(T)$

1: GetArrivalLists(T)$\rightarrow$ $A_J(T)$ and $A_D(T)$, see Section 5.1.2.
2: $i := 1$, $j := 1$, $t_{aJ} := 0$, $t_{aD} := 0$
3: **while** $i \leqslant h$ or $j \leqslant k$ **do**
4: **if** $\Delta t_J(i) \geqslant \Delta t_D(j)$ **then**
5: $t_{aD} := t_{aD} + \Delta t_D(j)$
6: GetDisjoinID $\rightarrow$ ID_D, see Section 5.1.3.
7: Add (D, ID_D, t_{aD}) into $RRL(T)$
8: $j := j + 1$
9: **else**
10: $t_{aJ} := t_{aJ} + \Delta t_J(i)$
11: GetJoinID $\rightarrow$ ID_J, see Section 5.1.3.
12: Add (J, ID_J, t_{aJ}) into $RRL(T)$
13: $i := i + 1$
14: **end if**
15: **end while**
16: Sort $RRL(T)$ according to increasing arrival times.
17: **return** $RRL(T)$

This means that during the given interval 2 join requests and 3 disjoin requests are collected, i.e. $h = 2$, $k = 3$. In addition, the requests feature the following inter-arrival times: $\Delta t_J(1) = 10$, $\Delta t_J(2) = 25$, $\Delta t_D(1) = 11$, $\Delta t_D(2) = 5$, and $\Delta t_D(3) = 7$.

In the first run of the while loop, the if-condition in Algorithm 1 is false because $\Delta t_J(1) < \Delta t_D(1)$. Therefore, the first join request is processed by determining its arrival time as $t_{aJ} := 0 + 10 = 10$, as $t_{aJ} = 0$ initially. Assuming that executing the process GetJoinID returned a member identity $ID_J = 5$, a first entry is written into the rekeying request list $RRL(T)$, as depicted in the first row of Table 4 which represents the $RRL(T)$ for this example. In the second iteration the if-condition is true because $\Delta t_J(2) > \Delta t_D(1)$. Therefore, the next request to be written to the $RRL(T)$ is of a disjoin type and has an arrival time $t_{aD} := 0 + 11 = 11$, as $t_{aD} = 0$ initially. Assuming that GetDisjoinID returns an ID_D which is equal to 3, the $RRL(T)$ is extended by the second entry of Table 4. The other entries of Table 4 can be determined in the same way. Figure 5 illustrates the relation between the inter-arrival times generated by the process GetArrivalList(T) and the estimated arrival times in the given example.

Request Type	**Member Identity**	**Arrival Time (ms)**
J	5	10
D	3	11
D	1	16
D	4	23
J	1	35

Table 4: $RRL(T)$ of Example 2

5.1.2. Arrival Process (GetArrivalLists)

Based on related work on modeling multicast member dynamics [15], the rekeying simulator assumes inter-arrival times, which follow an exponential distribution for join and disjoin requests with the rates λ and μ respectively. The corresponding cumulative distribution functions are given by:

$$F_J(\Delta t_J) = 1 - e^{-\lambda \Delta t_J} \qquad F_D(\Delta t_D) = 1 - e^{-\mu \Delta t_D} \tag{5}$$

To generate an exponentially distributed random variate based on uniform random numbers between 0 and 1, the inverse transformation technique can be used. Accordingly, if r represents such a random number, the inter-arrival times of a join and disjoin request can be estimated as:

$$\Delta t_J = -\frac{1}{\lambda} \ln r \qquad \Delta t_D = -\frac{1}{\mu} \ln r \tag{6}$$

Algorithm 2 outlines the arrival process for creating the join arrival list $A_J(T)$. Creating $A_D(T)$ is identical and omitted, for brevity.

5.1.3. Join/Disjoin Identity Selection Processes (GetJoinID/GetDisjoinID)

To join a member, any identity ID_J may be selected from the potential multicast subgroup M_D. A possible selection strategy may rely on choos-

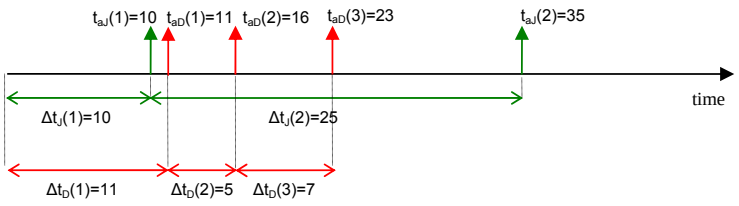


Figure 5: Arrival times and inter-arrival times for Example 2

Algorithm 2 GetArrivalLists

Require: T

Ensure: $A_J(T)$ (Generating $A_D(T)$ is identical)

1: $\sum \Delta t_J := 0$; $\sum \Delta t_D := 0$;
2: **while** $\sum \Delta t_J \leqslant T$ **do**
3: Generate r
4: Determine $\Delta t_J = 0$ according to (8)
5: $\sum \Delta t_J = \sum \Delta t_J + \Delta t_J$
6: Add Δt_J to $A_J(T)$
7: **end while**
8: **return** $A_J(T)$

ing the smallest available ID_J, which allows some order in the group management. In contrast, selecting a leaving ID_D from M_J is inherently nondeterministic, as a group owner can not forecast which member will leave the group. To select an ID_D the following method is proposed. The ID_D's of M_J are associated with continuous successive indices from 0 to $m-1$, where m is the number of all ID_D's in M_J. To select an ID_D, first a uniform zero-one random number r is generated. Then an index i is determined as $m \cdot r$. In a last step, the ID_D is selected, which has the index i.

5.2. Algorithm Manager

The algorithm manager plays a central role in the benchmark architecture. Its functionality can be illustrated by the process described in Figure 6. After reading the user settings of the desired parameters, the simulation mode, and the algorithms to be evaluated, the algorithm manager executes

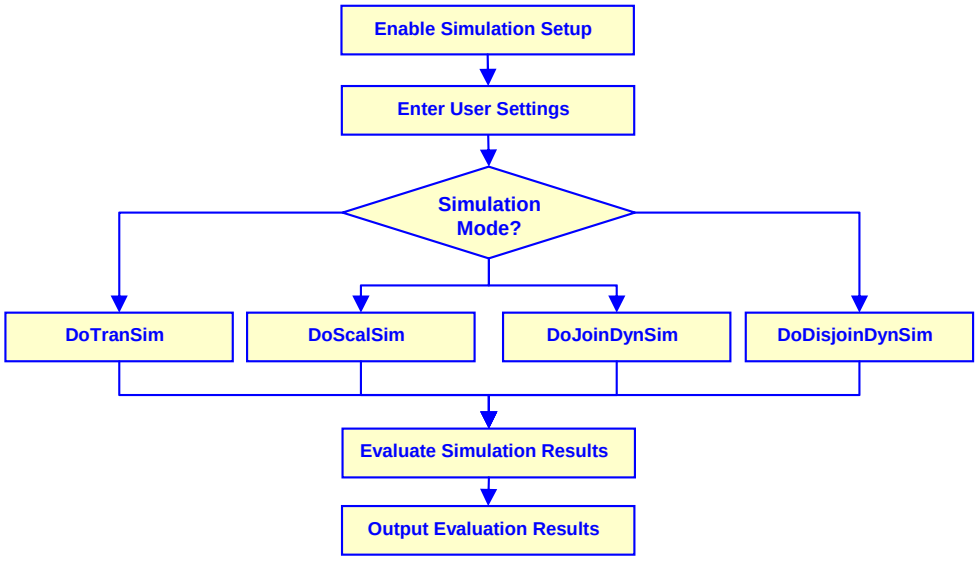


Figure 6: Rekeying performance evaluation procedure

the corresponding simulation process. Simulation processes on their part call the request generator and pass the rekeying requests to the selected rekeying algorithms. As a result, a simulation process provides abstract rekeying costs, see Definition 13. The abstract rekeying costs are then forwarded to the performance evaluator which determines the rekeying performance metrics JRT and DRT. In this section the underlying simulation processes DoTranSim, DoScalSim, DoJoinDynSim, and DoDisjoinDynSim are explained. For this purpose, some new concepts are introduced first.

Definition 13. ***Abstract Rekeying Cost (ARC)***

Abstract rekeying cost is a 9-tuple $(N_s(G), N_s(E), N_s(H), N_s(M), N_s(S), N_m(D), N_m(H), N_m(M), N_m(S))$, which specifies a rekeying request and gives the number of cryptographic operations executed by the server to grant this request as well as the the number of cryptographic operations executed by

the member to determine the new group key from the rekeying message. Table 5 specifies the elements of the ARC.

ARC Element	Meaning
$N_s(G)$	# Cryptographic keys generated on server
$N_s(E)$	# Symmetric encryptions executed on server
$N_s(H)$	# Cryptographic hash operations executed on server
$N_s(M)$	# Message authentication codes (MAC's) determined on server
$N_s(S)$	# Digital signatures determined on server
$N_m(D)$	# Symmetric decryptions executed by member
$N_m(H)$	# Cryptographic hash operations executed by member
$N_m(M)$	# MAC's verified by member
$N_m(S)$	# Digital signatures verified by member

Table 5: Abstract rekeying costs notation

Definition 14. ***Rekeying Cost List RCL(T)***

A rekeying cost list is a rekeying request list $RRL(T)$, see Definition 6, which is extended by the abstract rekeying cost ARC for each request. $RCL(T)$ is used for transient simulation.

***Example* 3.** Table 6 shows an example for an $RCL(T)$, which is an extension of the rekeying request list given in Table 3. This example results from executing the LKH algorithm with binary trees. This can be seen from the fact that each generated key is encrypted twice to determine the rekeying submessages. Note that the rekeying algorithm in this example does not use group authentication. Therefore, no message authentication codes are needed. Instead, rekeying submessages are hashed and the final hash value is signed once for each request [18].

Definition 15. ***Complex Rekeying Cost List CRCL(T)***

A complex rekeying cost list over an interval T is a set of rekeying cost lists

Request Type	Member Identity	Arrival Time (ms)	Rekeying Cost List RCL(T)								
			$N_s(G)$	$N_s(E)$	$N_s(H)$	$N_s(M)$	$N_s(S)$	$N_c(D)$	$N_c(H)$	$N_c(M)$	$N_c(S)$
L	1099	0	6	12	12	0	1	12	12	0	1
J	50	0.1	3	6	6	0	1	6	6	0	1
J	178	2	8	16	16	0	1	16	16	0	1
L	22657	5.3	2	4	4	0	1	4	4	0	1

Table 6: RCL(T) for Example 3

generated over this interval under different group conditions: $CRCL(T) = \{RCL1(T), RCL2(T), \cdots\}$.

$CRCL(T)$ is used in the simulation modes scalability, join dynamics, and disjoin dynamics, where an $RCL(T)$ is generated for each n, λ, or μ in the desired simulation range, respectively.

Transient Simulation. Algorithm 3 represents the process of transient simulation DoTranSim. The request generator process is resumed to generate a request list $RRL(t_{sim})$ for the desired simulation time period. For each selected rekeying algorithm, the algorithm manger performs two main steps. First, the rekeying algorithm is requested to construct an initial group with n_0 members. Then, each request of $RRL(t_{sim})$ is sent to the rekeying algorithm, which determines the corresponding abstract rekeying cost ARC for that request.

Other Simulation Modes. As mentioned in Section 4.2, other simulation modes are highly similar and rely all on the transient simulation mode. Therefore, only the scalability simulation is given in Algorithm 4, for brevity.

5.3. Performance Evaluator

This component receives a set of $RCL(T)$ or $CRCL(T)$ and calculates the rekeying performance metrics JRT and DRT as a function of time, group

Algorithm 3 DoTranSim

Require: All settings for a transient simulation as given in Table 2; set of rekeying algorithms to be evaluated.

Ensure: A $RCL(t_{sim})$ for each rekeying algorithm

1: GenReqList(t_{sim}) according to Algorithm 1 $\rightarrow$ $RRL(t_{sim})$
2: **for** each algorithm **do**
3: Initialize the group with n_0 members
4: **while** $RRL(t_{sim})$ is not empty **do**
5: Send a rekeying request to the algorithm
6: Get corresponding ARC
7: Add ARC to $RCL(t_{sim})$
8: **end while**
9: **end for**
10: **return** $RCL(t_{sim})$ for all algorithms

Algorithm 4 DoScalSim

Require: All settings for a scalability simulation as given in Table 2; set of rekeying algorithms to be evaluated.

Ensure: A $CRCL(T_o)$ for each rekeying algorithm

1: **for** each algorithm **do**
2: $n := n_{start}$
3: **while** $n \leqslant n_{end}$ **do**
4: DoTranSim for T_o and $n_0 = n$ according to Algorithm 3 $\rightarrow RCL(T_o)$
5: Add $RCL(T_o)$ to $CRCL(T_o)$
6: $n := n + \Delta n$
7: **end while**
8: **end for**
9: **return** $CRCL(T_o)$ for all algorithms

size, join rate, or disjoin rate.

Definition 16. ***Performance Simulation Point (PSP)***

A performance simulation point is a 3-tuple (x, JRT, DRT), where x can be time, n, λ, or μ depending on the simulation mode, see Table 2. Note that in a transient simulation JRT is not defined for a disjoin request. The same applies to DRT regarding a join request.

Definition 17. ***Rekeying Performance List (RPL)***

A rekeying performance list is a set of performance simulation points. $RPL = \{PSP\} = \{(x_1, JRT_1, DRT_1), (x_2, JRT_2, DRT_2), \ldots\}$.

Definition 18. ***Timing Parameter List (TPL)***

A timing parameter list (TPL) is a 9-tuple $(T_s(G), T_s(E), T_s(H), T_s(M), T_s(S), T_m(D), T_m(H), T_m(M), T_m(S))$, where the tuple elements specify the computational overhead for cryptographic operations both by the server and by the client, see Table 7.

Definition 19. ***Message Parameter List (MPL)***

A message parameter list (MPL) is a 4-tuple $(L(E), L(H), L(M), L(S))$, where the tuple elements specify the bit length of each segment in the rekeying message, see Table 7.

The performance evaluator executes processes, which combine a rekeying cost list $RCL(T)$ or a complex rekeying cost list $CRCL(T)$ with the timing and the message parameter lists TPL and MPL to produce a rekeying performance list PRL for a specific rekeying algorithm.

TPL/MPL Element	Meaning
$T_s(G)$	Time for generating one cryptograpic key on server
$T_s(E)$	Time for performing one symmetric encryption on server
$T_s(H)$	Time for performing one hash operation on server
$T_s(M)$	Time for detemining MAC on server
$T_s(S)$	Time for detemining a digital signature on server
$T_m(D)$	Time for performing one symmetric decryption by member
$T_m(H)$	Time for performing one hash operation by member
$T_m(M)$	Time for detemining MAC by member
$T_m(S)$	Time for verifying a digital signature by member
$L(E)$	Length of an encrypted key
$L(H)$	Lngth of a hash value
$L(M)$	Length of a MAC
$L(S)$	Length of a digital signature

Table 7: Timing and message parameters

For each rekeying request in $RCL(T)/CRCL(T)$ the join/disjoin rekeying times JRT and DRT are determined according to equations (1) and (11). Due to similarity, we only detail JRT in the following.

waiting time $t_w^{j/d}$. The waiting time for a join request is determined as:

$$t_w^{j/d} = \begin{cases} \sum_{i=1}^{m} t_{comp,server}^{j/d}(i) & \text{if } m \geqslant 1 \\ 0 & \text{if } m = 0 \end{cases} \tag{7}$$

where m represents the number of all requests waiting in the system queue or being processed at the arrival of the request at hand.

Server computing time $t_{com,server}^{j/d}$.

$$t_{comp,server}^{j/d} = N_s(G)\cdot T_s(G)+N_s(E)\cdot T_s(E)+N_s(H)\cdot T_s(H)+N_s(M)\cdot T_s(M)+N_s(S)\cdot T_s(S) \tag{8}$$

Client computing time $t_{com,server}^{j/d}$.

$$t_{comp,member}^{j/d} = N_m(D)\cdot T_m(D)+N_m(H)\cdot T_m(H)+N_m(M)\cdot T_m(M)+N_m(S)\cdot T_m(S) \tag{9}$$

Rekeying message delivery time $t_{comm}^{j/d}$.

$$t_{comm}^{j/d} = (N(E)\cdot L(E)+N(H)\cdot L(H)+N(M)\cdot L(M)+N(S)\cdot L(S))\times B \quad (10)$$

where B refers to the network bandwidth which can be entered as a simulation parameter to model the network traffic.

Transient Evaluation Process (EvalTranSimResults). In the case of a transient simulation the performance evaluator executes the process EvalTranSimResults according to Algorithm 5. For each join and disjoin request in the $JRT(t)$, a performance simulation point PSP is determined. The symbol ∞ in the pseudo code indicates an undefined metric for the current request. For example, JRT is not defined for a disjoin request. t_{aJ} and t_{aD} represent the arrival times of the corresponding join and disjoin requests, respectively. Remember that these time values are determined from the arrival lists by the request generator process according to Algorithm 1.

Complex Evaluation Process (EvalComplexSimResults). Other simulation modes deliver a $CRCL(T)$. The performance evaluator generates one performance simulation point PSP for each $RCL(T)$ of $CRCL(T)$. The first element of the PSP tuple represents a n, λ, or μ value for scalability, join dynamics or disjoin dynamics simulation, respectively. The second element represents the maximum join rekeying time JRT_{max} of all join requests in the observation time for the corresponding n, λ, or μ value. Similarly, the third element represents DRT_{max} of all disjoin requests. Algorithm 6 depicts the process EvalComplexSimResults for evaluating non-transient simulation results.

Algorithm 5 EvalTranSimResults

Require: A $RCL(t_{sim})$ for each rekeying algorithm

Ensure: A PRL for each rekeying algorithm

```
1: for each RCL(t_sim) do
2:    for each request in RCL(t_sim) do
3:       if request type = J then
4:          Determine JRT according to equations (1), (7),(8), (9), and (10)
5:          PSP=(t_aJ, JRT, ∞)
6:       else
7:          Determine DRT according to equations (11), (7),(8), (9), and (10)
8:          PSP=(t_aD, ∞, DRT)
9:       end if
10:      Add PSP to PRL
11:   end for
12: end for
13: return PRL for all algorithms
```

Algorithm 6 EvalComplexSimResults

Require: A $CRCL(T_o)$ for each rekeying algorithm

Ensure: A PRL for each rekeying algorithm

1: **for** each rekeying algorithm **do**
2: **for** each $RCL(T_o)$ of $CRCL(T_o)$ **do**
3: $JRT_{max} = 0$, $DRT_{max} = 0$
4: **for** each request in $RCL(T_o)$ **do**
5: **if** request type = J **then**
6: Determine JRT according to equations (1), (7),(8), (9), and (10)
7: **if** $JRT > JRT_{max}$ **then**
8: $JRT_{max} := JRT$
9: **end if**
10: **else**
11: Determine DRT according to equations (11), (7),(8), (9), and (10)
12: **if** $DRT > DRT_{max}$ **then**
13: $DRT_{max} := DRT$
14: **end if**
15: **end if**
16: PSP=($n/\lambda/\mu$, JRT_{max}, DRT_{max})
17: **end for**
18: Add PSP to PRL
19: **end for**
20: **end for**
21: **return** PRL for all algorithms

6. Implementation

The rekeying benchmark was implemented in Java using the Eclipse Environment [19]. The software architecture consists of two main components: the graphical user interface (GUI) and the actual simulation kernel, as depicted in Figure 7. In this figure the benchmark software is illustrated as a simplified class diagram according to the Unified Modeling Language (UML) [20]. Note assigning the different classes to two packages denoted as *gui* and *kernel*. *BenchmarkAndAlgorithmManager* represents the central class in the package kernel and includes the main function. This class is associated with the class *SimulationSettings*, which receives its attribute values from the GUI class *SimulationSetup*.

The execution of the benchmark causes the opening of a framework, where several simulations can be executed. This point is indicated by the association relation between the classes *SimulationSetup* and *MainFrame*. *Simulation* is an abstract class, which is inherited by two different simulation classes: *TransientSimulation* and *ComplexSimulation*. Note that the class *ComplexSimulation* is also abstract and builds the base class for the other three simulation classes *ScalabilitySimulation*, *JoinDynSimulation*, and *DisjoinDynSimulation*. The association relation between the classes *TransientSimulation* and *ComplexSimulation* reflects the fact that each complex simulation is based on a frequented execution of the transient simulation. After program start, the simulation setup window displays default parameters. Changing these values is stored for a next simulation in the same session. See Figure 8 for an overview of the simulation setup window. The set of all parameters belonging to one simulation are managed as a parameter list

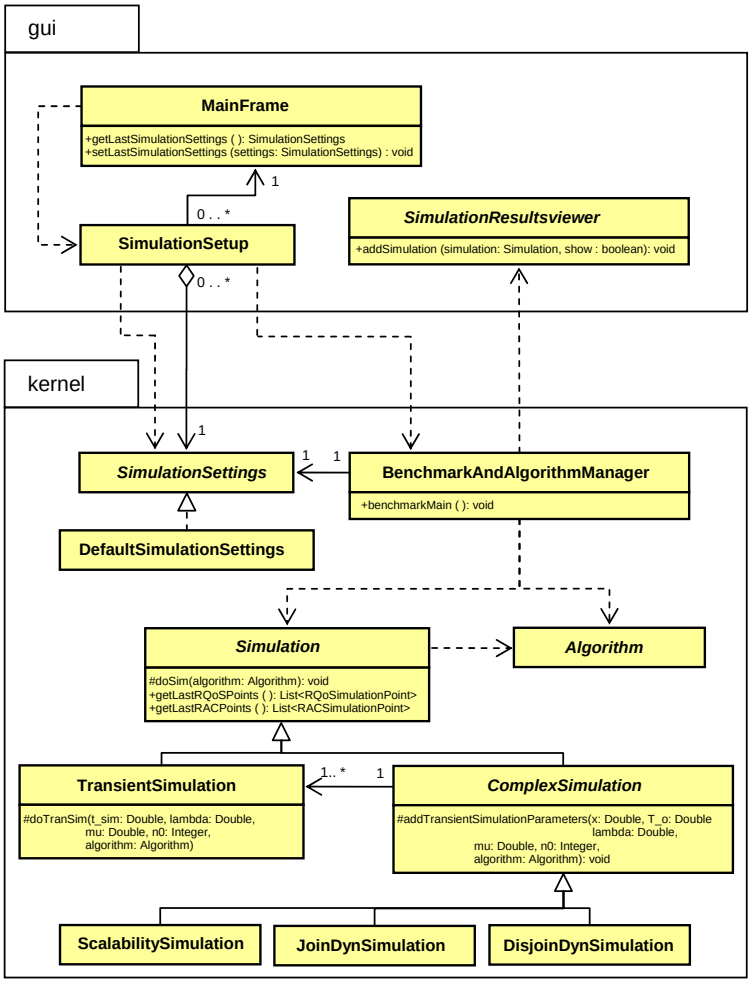


Figure 7: Rekeying benchmark class diagram

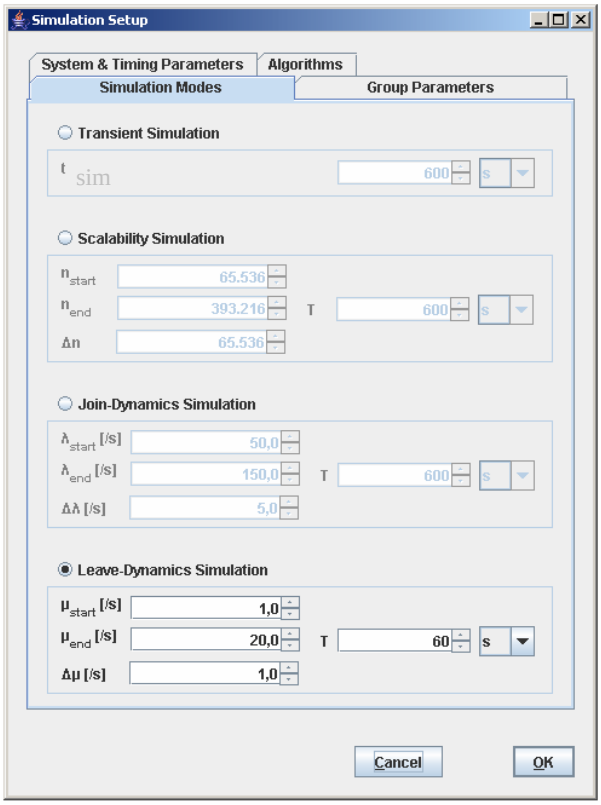


Figure 8: Pay-TV a potential scenario for secure multicast

using the library class *LinkedHashMap*. This class is not shown in Figure 7 for simplicity.

7. Case Studies

This section illustrates the advantage of the rekeying benchmark by means of two case studies. Both examples relate to the logical key hierarchy. Therefore, this scheme is introduced first.

The basic idea behind LKH is to divide the group into hierarchical subgroups and to provide the members of each subgroup with a *help-key*. Con-

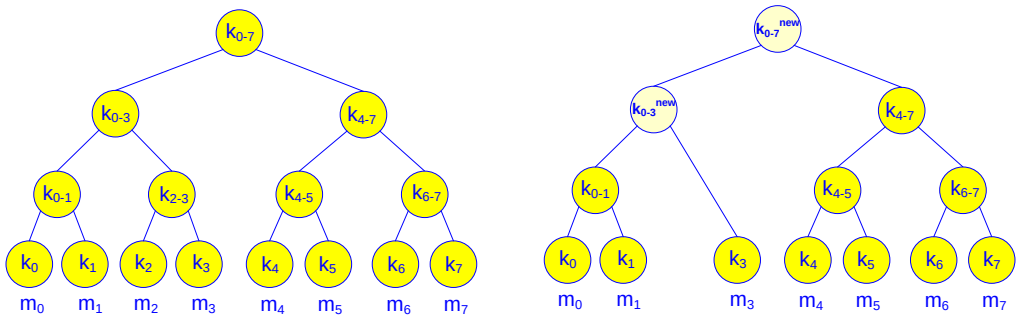


Figure 9: LKH example

sider the example illustrated in Figure 9 (left tree) for an eight-member group. In this model members m_0 and m_1, for instance, build a subgroup with the help-key k_{0-1}. All members compose the largest subgroup with the help-key k_{0-7}. This key represents the group key, which is used to encrypt payload data. Consequently, a member holds several keys, which are the identity key k_d, which is known only to this member and to the server, the group key k_g known to all group members, and some help-keys k_{x-y} corresponding to the subgroups, which the member belongs to. Member m_6, for example, holds $k_d = k_6$, $k_g = k_{0-7}$, and two help-keys, namely k_{6-7} and k_{4-7}.

To disjoin m_2, for instance, two keys, k_{0-3}^{new} and k_{0-7}^{new} are generated, encrypted, and sent to the remaining members needing them. The right side of Figure 9 represents the key tree after this processing. Using the notation $E_{k_a}(k_b)$ to refer to a *rekeying submessage* (RSM) representing the encryption of the key k_b with the key k_a, the server has to generate the following rekeying submessages in order to disjoin m_2:

$$E_{k_3}(k_{0-3}^{new}), E_{k_{0-1}}(k_{0-3}^{new}), E_{k_{0-3}^{new}}(k_{0-7}^{new}), E_{k_{4-7}}(k_{0-7}^{new})$$

A similar analysis can be performed for the case of member joining. LKH

enables efficient rekeying due to the logarithmic relation of rekeying cost to the group size.

7.1. Tree Rebalancing in LKH

LKH is based on the management of a key tree on the rekeying server. As an effect of multiple disjoin processes, the key tree may get out of balance. Several solutions have been proposed to rebalance the tree in this case. The first contribution originates from Moyer [11] who introduced two methods to rebalance the key tree, an immediate and a periodic rebalancing. Only a cost analysis after one disjoin request is given for the first method. The periodic rebalancing is not analyzed. Moharrum [12] presented a method for rebalancing based on sub-trees. A comparison with the solution of Moyer is drawn, but not with the original LKH. Rodeh [13] applied AVL-tree rebalancing methods to key trees. However, no backward access control is guaranteed in this solution. Goshi [14] proposed three algorithms for tree rebalancing. Simulation results are provided, which assume equally likely join and disjoin behavior. However, this condition itself ensures tree balancing, because a new member can be joined at the leaf of the most recently removed member. The same applies to the simulation results by Lu [9]. From this description, it is obvious that a comprehensive analysis is needed to justify the employment of rebalancing, which is associated with extra rekeying costs resulting from shifting members between tree leaves. The rekeying benchmark offers this possibility by allowing a simultaneous evaluation of two LKH algorithms (with and without rebalancing) under complex conditions. Especially the effect of the disjoin rate is of interest in case of rebalancing.

Therefore, a disjoin dynamics simulation is performed and parameters

particularly emphasizing the effects of rebalancing have been chosen. As members are joined and disjoined randomly during simulation, balancing cannot be triggered directly. For this reason the following considerations have been taken into account:

- High rate of leaving members for disturbing tree balance and promoting rebalancing: $\mu_{start} = 1s^{-1}$, $\mu_{stop} = 500s^{-1}$, $\Delta\mu = 25s^{-1}$, $T_o = 20s$
- Low rate of joining members for not affecting rebalancing process caused by leaving members: $\lambda = 5s^{-1}$
- High quantity of initial members compared to maximum group size leading to high rebalancing rate : $n_0 = 65535$, $n_{max} = 131072$

For aquiring meaningful measurement data also the timing values according to available implementations have been chosen. For the server side, values for primitive crypto operations of a rekeying processor introduced in [18] have been applied.

- 0.48 μs for AES-128 key generation
- 0.24 μs for encrypting a 128 Bit AES key
- 0.24 μs for computing a SHA-256 hash value of an 128 Bit AES key
- 0.24 μs for generating a HMAC-SHA1 message authentication code of an 128 Bit AES key

For determining the client side values of primitive crypto operations, the Flexiprovider library [21] on a PC (Windows Vista 32 Bit SP2 running on Pentium Core 2 Duo L7500@1.6 GHz, 2 GB RAM) has been used.

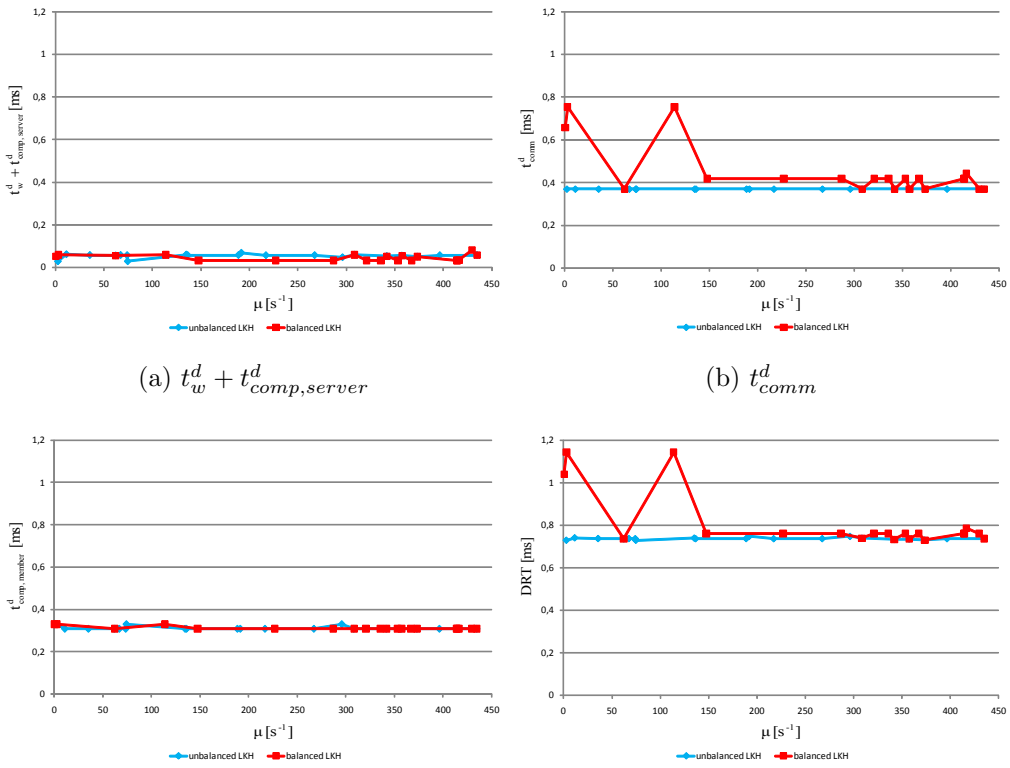
- 6.83 μs for decrypting an 128 Bit AES key
- 4.48 μs for for computing a SHA-256 hash value of an 128 Bit AES key
- 9.29 μs for verifying a HMAC-SHA1 message authentication code of an 128 Bit AES key

Simulation results are depicted in figure 10. Please remember the definition of DRT:

$$DRT = t^d_w + t^d_{comp,server} + t^d_{comm} + t^d_{comp,member} \tag{11}$$

7.2. Comparison of LKH with OFT

Oneway-Function-Trees(OFT) introduced by Sherman [22] are also binary trees used for efficient rekeying. Additionally to LKH a blinding function $g(x)$ and a mixing function $f(x)$ are used. The blinding function $g(x)$ is a one-way function, meaning that it is computationally infeasable to compute the value of x when $g(x)$ is given. The mixing function $f(x, y)$ merges two values x and y and fulfills the property $f : \mathbb{N}^n \times \mathbb{N}^n \rightarrow \mathbb{N}^n$. Usually the blinding function is implemented by using a hash function and the mixing function is implemented by using an XOR-function. Secret keys, which have been assigned beforehand over a secure channel and which are associated with the leaf nodes of the tree, represent the member keys. Applying the blinding function $g(x)$ on these member keys results in the blinded keys of the members. Each of the inner nodes is also associated with a blinded key and corresponding secret key, the key encryption key (KEK). The KEK of



(a) $t^d_w + t^d_{comp,server}$

(b) t^d_{comm}

(c) $t^d_{comp,member}$

(d) DRT

Figure 10: Leave dynamics simulation unbalanced vs. balanced LKH

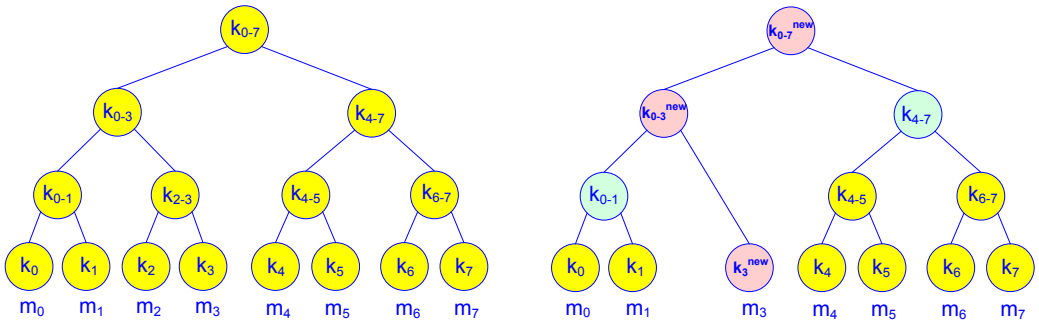


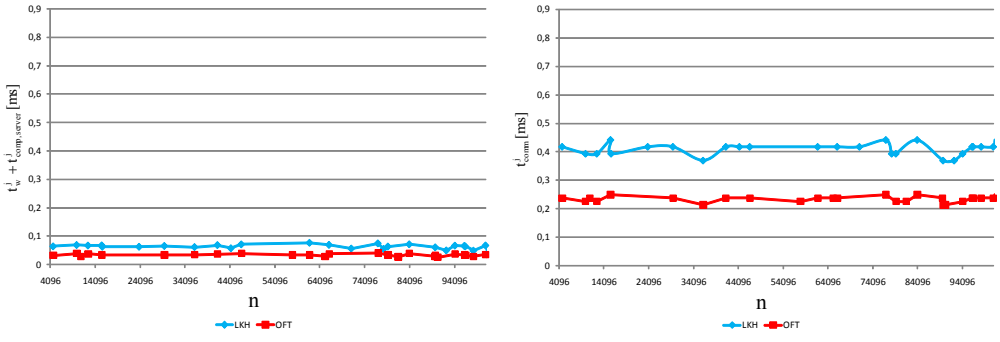
Figure 11: OFT example

an inner node is computed by applying the mixing function $f(x, y)$ on the blinded keys of its children. The blinding key of an inner node is computed by applying the blinding function $g(x)$ on its KEK. The OFT key distribution provides that each member knows his own member key and all blinded keys of the siblings of the respective inner nodes on the path from the leaf node to the root. Consider the example for an eight-member group illustrated in figure 11 again. When disjoining member m_2 (right tree) the key of its sibling m_3, k_3, is directly associated with the parent node k_{0-3}. After that k_3 is rekeyed and the new KEK k_3^{new} is sent by using the old KEK k_3. Subsequently by using the new blinded keys, a rekeying of all nodes in the tree from member m_3 to the root takes place, namely of k_{0-3} and k_{0-7}. Finally the siblings of the nodes, whose blinded keys have been changed, are provided with the new corresponding blinded keys; in our case the node of k_{0-1} is provided with $E_{k_{0-1}}(g(k_3^{new}))$ and the node of k_{4-7} is provided with $E_{k_{4-7}}(g(k_{0-3}^{new}))$. In Figure 11 the renewal of a KEK is indicated by the label in the respective node, the renewal of a blinded key is indicated by the red background color of the respective node and the renewal of the information

about the siblings' blinded key is indicated by the green background color of the respective node. Alltogether, the rekeying message for the disjoin operation of m_2 consists of the following rekeying submessages:

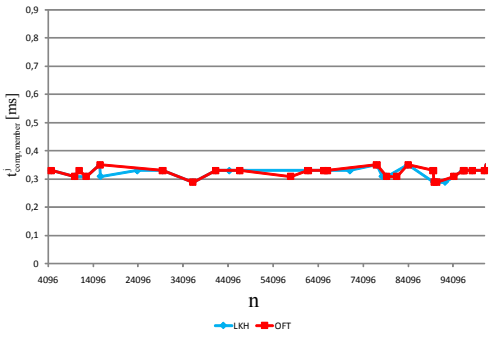
$$E_{k_3}(k_3^{new}), E_{k_{0-1}}(g(k_3^{new})), E_{k_{4-7}}(g(k_{0-3}^{new}))$$

The primary advantage of OFT over LKH is that it sends only half as many keys as LKH to the group and performs only half as many encryptions. This advantage comes for the cost that additional blinding function computations in the number of the tree height have to be performed for OFT. However, it is clear that the advantage of OFT is unbeatable especially in cases where line bandwidth is low and therefore communication costs are dominant. In figure 12a JRTs of LKH and OFT resulting from a scalability simulation with $n_{start} = 4096$, $n_{end} = 131.072$ and a bandwidth of 16 Mbit/s are depicted. The system, timing and message parameters are the same as in our tree balancing example. For the blinding function the timing values of the aforementioned SHA-256 hash function are used. It can be observed that even if line bandwidth as high as 16 Mbit/s is assumed, communication time is the dominant factor leading to better JRT performance of OFT. In cases where line speed does not matter, it could be useful to know how the computational cost ratio between the encryption function and the blinding function is in order to decide whether to use LKH or OFT. For instance when a particular environment of an embedded system is considered, in which the encryption function is available as a high-performance hardware implemention and the blinding-function is implemented in software. By using the rekeying benchmark it is possible to assess the performance of LKH and OFT in such an environment under various group conditions. In order to determine the com-

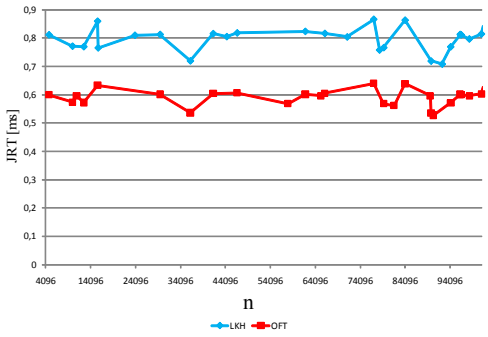


(a) $t_w^j + t_{comp,server}^j$

(b) t_{comm}^j



(c) $t_{comp,member}^j$



(d) JRT

Figure 12: Scalability simulation LKH vs. OFT

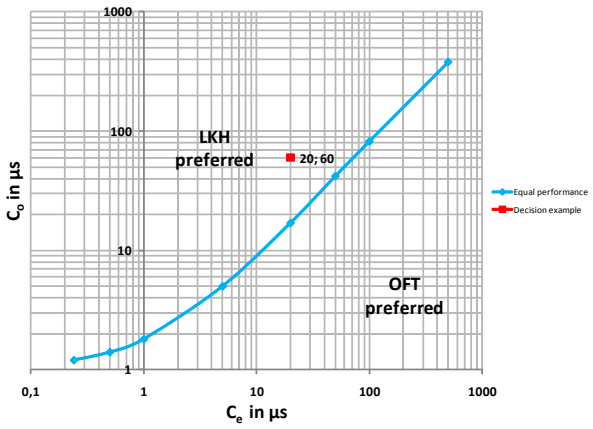


Figure 13: Equal performance of LKH and OFT depending on C_e and C_o

putational cost ratio the timing parameters of a single encryption operation C_e and a single blinding operation C_o were adjusted in such way that LKH and OFT performed equal in a scalability simulation under the conditions decribed above. Please note that communication costs have been omitted. The results are depicted in figure 13. Using these results, a dicision whether to use LKH or OFT in a particular environment can easily be made. For instance, in an environment, where C_e=20μs and C_o=60μs the use of LKH would be preferred (see figure 13). As a more general statement it can be observed that from the view of server side computation costs the advantages of OFT over LKH fade as costs for the encryption operation increase. This is due to the fact that OFTs also utilize the encryption function.

8. Conclusion

An assessment methodology and an associated simulation tool were presented as a novel method to deal with the rekeying performance evaluation

problem. By means of the underlying concept of abstraction a reliable and meaningful evaluation of different rekeying algorithms is provided. Two case studies illustrated the advantage of this benchmark in analyzing yet unanswered questions relating to rekeying. In its first prototype, the benchmark considers rekeying costs in terms of cryptographic operations to be run on the server side. Other cost factors, such as tree traversing in LKH, will be addressed in future work. Additionally, more rekeying algorithms will be programmed and evaluated.

References

[1] C.K. Wong, M. Gouda, and S.S. Lam, "Secure Group Communication Using Key Graph", IEEE/ACM Trans. on Networking, Vol. 8, No. 1, February 2000, pp. 16-30.

[2] W.H.D. Ng, and Z. Sun, "Multi-Layers Balanced LKH", in Proc. of IEEE Int. Conf. on Communication ICC, May 2005, pp. 1015-1019.

[3] X.S. Li, Y.R. Yang, M. Gouda, and S.S. Lam, "Batch Rekeying for Secure Group Communications", in Proc. ACM 10th Int. World Wide Web Conf., Hong Kong, May 2001.

[4] Y. Amir, Y. Kim, C. Nita-Rotaru, and G. Tsudik, "On the Performance of Group Key Agreement Protocols", ACM Trans. on Information Systems Security, 7(3), 2004, pp. 457-488.

[5] J. Pegueroles, and F. Rico-Novella, "Balanced Batch LKH: New Proposal, Implementation and Performance Evaluation", in Proc. IEEE Symp. on Computers and Communications, 2003, pp. 815.

[6] W. Chen, and L.R. Dondeti, "Performance Comparison of Stateful and Stateless Group Rekeying Algorithms", in Proc. of Int. Workshop in Networked Group Communication, 2002.

[7] A. Sherman, and D. McGrew, "Key Establishment in Large Dynamic Groups Using One-Way Function Trees", IEEE Trans. on Software Engineering, Vol. 29, No. 5, May 2003, pp. 444-458.

[8] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, "The VersaKey Framework: Versatile Group Key Management", IEEE J. on Selected Areas in Communications, Vol. 17, No. 8, August 1999, pp. 1614-1631.

[9] H. Lu, "A Novel High-Order Tree for Secure Multicast Key Management", IEEE Trans. On Computers, Vol. 54, No. 2, February 2005, pp. 214-224.

[10] S. Mittra, "Iolus: A Framework for Scalable Secure Multicasting", in Proc. of ACM SIGCOMM, Cannes, France, September 1997, pp. 277-288.

[11] M.J. Moyer, G. Tech, J.R. Rao, and P. Rohatgi, "Maintaining Balanced Key Trees for Secure Multicast", Internet draft, June, 1999, http://www.securemulticast.org/draft-irtf-smug-key-tree-balance-00.txt.

[12] M. Moharrum, R. Mukkamala, and M. Eltoweissy, "Efficient Secure Multicast with Well-Populated Multicast Key Trees", in Proc. of IEEE ICPADS, July 2004, pp. 214.

[13] O. Rodeh, K.P. Birman, and D. Dolev, "Using AVL Trees for Fault Tolerant Group Key Management", Tech. Rep. 2000-1823, Cornell University, 2000.

[14] J. Goshi, and R.E. Ladner, "Algorithms for Dynamic Multicast Key Distribution Trees", in Proc. of ACM Symp. on Principles of Distributed Computing, 2003, pp. 243-251.

[15] K.C. Almeroth, and M.H. Ammar, "Collecting and Modeling the join/leave Behavior of Multicast Group Members in the MBone" in Proc. of HPDC, 1996, pp. 209-216.

[16] NIST (National Institute of Standards and Technology), "Advanced Encryption Standard (AES)", Federal Information Processing Standard 197, Nov. 2001.

[17] http://www.cryptopp.com/

[18] Shoufan A., Laue R., and Huss S.A., "High-Flexibility Rekeying Processor for Key Management in Secure Multicast" IEEE Int. Symposium on Embedded Computing SEC-07, Niagara Falls, Canada, May 2007.

[19] Eclipse, Open Development Platform, Version 3.1.2, http://www.eclipse.org/

[20] Kecher C., "UML 2.0", Galileo Computing, 2005.

[21] Flexiprovider, Toolkit for the Java Cryptography Architecture, Version 1.6p7, http://www.flexiprovider.de/

[22] Balenson, D., D. McGrew, and A.Sherman, "Key management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization", draft-ietf-smug-groupkeymgmt-oft-00.txt, IRTF August 2000, work in progress