Secure Multicast for Virtual Private Converencing

Clemens Schroedter

Master Thesis, June 4, 2004

Supervisor:
Reto Strobl
IBM Zurich Reasearch Laboratory

Master Professor:
Prof. Dr. Roger Wattenhofer
Distributed Computing Group, ETH Zurich

Contents

1 Introduction

Multicast allows a source to disseminate data among multiple recipients, called the *multicast group*, such that the actual data packages are routed efficiently over the physical network links. *Secure* multicast additionally provides authenticity and privacy of the multicast data. Authenticity means that each group member can recognize whether a message was sent by a particular source, whereas secrecy means that only the multicast group members (and all of them) learn the transmitted data.

Multicast is at the core for many collaborative applications. Examples include Internet video transmissions, stock quotes, news feeds, software updates, live multi-party conferencing, on-line video games and shared white-boards. By the diversity of these scenarios, it soon becomes clear that there is little hope for a unified security solution that accommodates all scenarios. Therefore, Canetti et al. [CGI+99] suggest to investigate security solutions for two 'bench-mark' scenarios, which are not only important on their own, but also have the property that solutions for these scenarios may serve as a good basis for other scenarios. The first scenario involves a single sender (e.g. an on-line stock-quotes distributor) and a large number of recipients (e.g. hundres of thousands). The second scenario — which we consider in this work — is on-line virtual private conferencing among up to a few hundreds of participants, where many (or all) of the members may be sending data to the group.

The standard approach for implementing secure multicast in the second scenario is to provide all members of the group (and only the members) with a secret *group key* under which all messages to be multicast are encrypted. Authenticity is usually achieved through digital signatures, or some form of symmetric message authentication codes (MACs) [CGI+99]. Thus, setting up secure multicast for private virtual conferencing essentially reduces to the problem of generating and distributing a secret group key. This can be done through a so-called *group key exchange* (GKE) protocol, which allows a group of servers to compute a secret key that remains well hidden from anyone outside the group. The problem with using such protocols is that they only terminate if all group members participate. Thus, if the group key is to be used for secure multicasting in a virtual conference scenario, the secure communication will only become available once the last group member has joined the conference. This is certainly undesirable, as in many practical scenarios, members may be joining at different points in time, or not join the conference at all. Servers that join the conference early should be able to compute the key without having to wait for *all* servers to join the conference.

One way to solve this problem is to integrate GKE protocols with a view-based group communication system (GCS), that provides the abstraction of "currently live nodes" to all servers of the multicast group in a consistent way. The idea is now to run a GKE protocol only among those members of the multicast group that are reported to be live by the GCS. Whenever a membership change occurs, i.e., when a member joins or leaves the conference, the GKE protocol is run from scratch among the new set of live servers. This way, early joining members will be able to communicate before all servers have joined the system. The disadvantage of this approach is that GCSs rely on timeouts to detect membership events, and thus are subject to timing attacks, which are often easy to launch in practice.

In order to avoid these issues, we suggest to run a *fault-tolerant* GKE protocol among the *entire* multicast group, instead of integrating standard GKE protocols with a GCS. A fault-tolerant GKE protocol terminates for all servers as long as a majority of the participants remain up. Thus, if the key is to be used for private virtual conferencing, secure multicast communication will become available as soon as a majority of the members have joined the conference. The advantage of this approach is that it is completely asynchronous and therefore robust against timing attacks. Furthermore, it facilitates key management, as there is only a single group key being generated. On the downside, it can only be applied in conference scenarios where a majority of group members are guaranteed to join the conference.

In the first part of this work, we present a performance evaluation of a secure multicast implementation that uses the fault-tolerant GKE protocol proposed by Cachin and Strobl [CS04] for group key generation, and digital signatures for authentication. In the second part of this work, we describe the architecture of `SecChat`, which is a peer-to-peer application allowing peers to dynamically setup virtual conferences among various groups of peers without relying on a central component. `SecChat` uses an (untrusted) directory service for storing information on conferences to be held, such as the group members, or the date and time of the conference. It allows any peer to announce a conference by storing the corresponding entry in the directory. In order to achieve secure and efficient multicast communication among members of a conference, `SecChat` uses the solution for secure multicast we described before. This yields a system for secure on-line virtual conferencing where security is guaranteed as soon as a majority of the conference members have entered a conference.

The thesis is organized as follows. In Section 2, we describe our system model, and our construction of a secure multicast channel from an agreement-based GKE protocol. In Section 3, we present our experimental results on the performance of the construction. Finally, in Section 4, we describe the architecture and implementation of `SecChat`.

2 Secure Multicasting among Majorities

We now describe our protocol for secure multicast. Our construction consists of three protocols: a fault-tolerant consensus protocol, a fault-tolerant GKE protocol, and a protocol for secure multicast. The consensus protocol serves as sub-protocol for the GKE protocol, which in turn serves as sub-protocol for the secure multicast protocol. In the following, we describe our system model and provide a detailed description of the protocols.

2.1 The System Model

We consider a multicast group of n servers $MC = \{S_1, \dots, S_n\}$ that wish to establish a secure multicast channel for private virtual conferencing. We assume that at most a minority of these servers *crash*, i.e., stop executing the protocol at some point. We call such servers *crashed*, and all other servers *honest*.

The servers in MC are connected by reliable asynchronous point-to-point links and have no access to a common clock. Every server $S_i \in MC$ has a public/private signature key pair (p_i, s_i), and knows the public keys of all other servers. The point-to-point links are authenticated using digital signatures and message authentication codes (MAC). Specifically, the first time two servers wish to communicate with each other, they exchange a symmetric key using a standard two-party key exchange protocol [BCK98, Sho99, BPR00]. During this stage, digital signatures (under the distributed public keys) are used to authenticate the point-to-point link. Once the symmetric key is established, the point-to-point link is authenticated using a message authentication code (MAC) under the established key.

Apart from the point-to-point links, every server can also send messages to all other servers through a reliable multicast channel. This channel neither provides authenticity nor secrecy of transmitted messages, and is completely asynchronous, i.e., messages may be delayed arbitrarily.

We will state the theoretical complexity of a protocol in this setting in terms of the total number of messages sent by honest servers, the size of these messages in bits, and the number of sequential message exchanges, called *asynchronous rounds*, that it takes until the first server may terminate the protocol.

2.2 Asynchronous Consensus with Failures

An important primitive of our solution for GKE is a fault-tolerant consensus protocol. In a consensus protocol, every server receives as input at bit string of some length L, and produces as output some bit string of length L. The goal is that all servers output the same bit string, and that this bit string corresponds to the input of at least one servers. Furthermore, every server must terminate, provided that a majority of the servers invoke the protocol and do not crash. In the protocol specification below, we allow the servers to invoke several instances of a consensus protocol concurrently, where every instance is identified by a tag ID. The tag may be an arbitrary bit string.

Specification. Technically, a consensus protocol has the following interface. Every server S_i accepts input actions of the form $(\mathsf{propose}, ID, v)$. If this happens, we say S_i *proposes* v *for (instance)* ID. Every server S_i produces output actions of the form $(\mathsf{decide}, ID, \bar{v})$. If this happens, we say S_i *decides* $\bar{v}$ *in (instance)* ID . With respect to this interface, a fault-tolerant consensus protocol satisfies the following properties:

TERMINATION: If $\lceil \frac{n+1}{2} \rceil$ honest servers propose some value for an instance ID, then all honest servers that propose a value for ID eventually decide some value in instance ID.

AGREEMENT: If two servers decide values v_i and v_j in an instance ID, then $v_i = v_j$.

VALIDITY: If some server decides a value v_i in an instance ID, then at least one server has proposed v_i for ID before.

The Protocol. Fischer et al. [FLP85] showed that there exists no *deterministic* asynchronous fault-tolerant consensus protocol. To circumvent this impossibility result, one can either use *randomized* protocols, or extend the system model by some form of timing assumption (see [Asp02] for a survey of the currently known asynchronous consensus protocols of either type). The most efficient randomized consensus protocols are asymptotically constant-round; see for example Canetti and Rabin's construction [CR93]. However, these protocols are not very efficient in terms of their concrete complexity, and only allow to agree on binary values; it is not immediately clear how to derive consensus protocols thereof that allow to agree on bit strings of arbitrary length, as we will need it.

In this work, we will therefore use a consensus protocol due to Chandra and Toueg [CT96] that relies on so-called *failure detectors* [CT96]. A failure detector is a local module available to every server that periodically outputs a list of servers that it suspects to have crashed and is usually based on a timing assumption. The consensus protocol that we will use relies on a failure detector $\diamond S$ that satisfies the following two properties:

EVENTUAL WEAK ACCURACY: There is an honest server and a time after which that server is not suspected to have crashed by the failure detector of any other honest server.

EVENTUAL STRONG COMPLETENESS: There is a time after which every crashed server is permanently suspected by the failure detector of any honest server.

To implement $\diamond S$, we use a protocol due to [CT96]. It assumes that every server has a local clock, and relies on the following synchrony assumption: there are bounds on the relative speeds of the local clocks and on message transmission times, but these bounds are not known *and* they hold only after some unknown time. These assumptions are also known as *partial synchrony*, and denoted by $\mathcal{M}_3$. The implementation for $\diamond S$ works as follows.

Each server S_i periodically sends a "S_i-is-alive" message to all other servers. If S_i does not receive a "S_j-is-alive" message from some server S_j for $\Delta_{S_i}(S_j)$ time units on its local clock, S_i adds S_j to its list of suspects. If S_j receives "S_j-is-alive" from some process S_j that it currently suspects, S_i knows that its previous time-out on S_j was premature. In this case, S_i removes S_j from its list of suspects and increases its time-out period $\Delta_{S_i}(S_j)$. For a more detailed description of this protocol, we refer to [CT96].

We now proceed with the description of the fault-tolerant consensus protocol that we use in this work. The protocol uses the *rotating coordinator* paradigm [Rei82]; the idea is to proceed in "rounds" as follows (these are different than the asynchronous rounds we consider for stating the efficiency of a protocol). During a round r, the current coordinator is server S_c for $c = (r \mod n) + 1$. All messages are either to or from the current coordinator. The current coordinator S_c tries to determine a consistent decision value. If S_c is honest and is not suspected by any other honest server, then the coordinator will succeed, and it will broadcast this decision value. Otherwise, the servers proceed with the next round.

Every round of this consensus protocol is divided into the following four asynchronous phases.

Phase 1: Every server sends its current estimate $\mathit{estimate}_i$ of the decision value timestamped with the round ts_i number in which it adopted this estimate, to the current coordinator S_c. In the first round, $\mathit{estimate}_i$ is the input of server i, and ts_i is set to 1.

Phase 2: The coordinator waits for receiving $\lceil \frac{n+1}{2} \rceil$ such estimates, selects one with the largest time stamp, and sends it to all the servers as their new estimate $\mathit{estimate}_c$

Phase 3: Every server S_i waits for either receiving $\mathit{estimate}_c$ from S_c, or for receiving a signal from its failure detector indicating S_c has crashed.

In the first case, S_i adopts $\mathit{estimate}_c$ as its new estimate, and sends *ack* to the coordinator. In the second case, S_i sends *nack* to S_c.

Phase 4: The coordinator waits for $\lceil \frac{n+1}{2} \rceil$ replies (*ack* s or *nack* s). If all replies are *ack* s, then S_c sends a request to decide on $\mathit{estimate}_c$ to all other servers.

At any time, if a server receives such a request, it echos the request to all other servers. A server decides $estimate_c$ and terminates the protocol if it either receives the direct request, or an echo thereof.

It is easy to see that if after some time, there exists an honest server that is not suspected by any other honest server, then this server will succeed to determine a consistent decision value in the round where it is the coordinator, and the protocol will terminate.

Validity of the protocol follows immediately, as only values are proposed as estimates that at least one server has received as input. To see agreement, note that a coordinator only requests to decide on a value $estimate_c$ if a majority of servers have adopted $estimate_c$ as their current estimate. The way a coordinator selects the estimate it proposes ensures that in any subsequent round, the current coordinator will propose the same value $estimate_c$ as the new estimate.

We remark that the protocol is not fully asynchronous as it relies on a failure detector. However, as pointed out by [CT96], it is only the termination property of the protocol that depends on the timing assumption underlying the failure detector. Specifically, invalidating this timing assumption only *delays* the protocol, but does not affect its agreement or validity property.

Complexity. The efficiency of the protocol depends on the number of rounds the servers go through until the failure detectors stabilize, i.e., until there exists an honest server that is not suspected to have crashed by all other honest servers. Once this happens, this coordinator will fix the decision value, and the servers will terminate. Assuming that the failure detectors stabilize after R rounds, then the protocol uses $3Rn+n^2$ messages of size L bits in $4R$ rounds.

2.3 Asynchronous Group Key Exchange with Failures

An important primitive of our solution for secure multicasting is a fault-tolerant GKE protocol. Such a protocol allows the servers of the multicast group to repeatedly establish a secret group key, where every such key is identified by a unique tag $ID \in \{0,1\}^*$. The protocol ensures that the key remains hidden from anyone outside the group that can only observe the network traffic. Moreover, it guarantees that if at least a majority of servers initialize the generation of a key with tag ID, then all servers eventually terminate and compute the key.

Specification. More technically, a fault-tolerant GKE protocol has the following interface. Every server S_i accepts input actions of the form (init, ID), and produces output actions of the form $(\mathsf{finished}, ID, sk)$. If a server S_i receives such an input, we say S_i *initializes session ID* , and if it produces such an output, we say S_i *computes key sk in session ID* . With respect to this interface, a fault-tolerant GKE protocol satisfies the following properties:

Termination: If a majority of servers initialize a session ID, then all servers that initialize ID eventually compute a key sk in session ID.

Secrecy: Only the servers of the multicast group learn the keys being computed.

The Protocol. In this work, we will use an implementation for fault-tolerant GKE recently proposed by Cachin and Strobl [CS04]. It is the first known implementation for fault-tolerant GKE, and assumes a setting where the servers are connected by authenticated point-to-point links. The protocol uses a public key encryption scheme, and relies on a fault-tolerant sub-protocol for asynchronous consensus. Given these primitives, protocol `GKE` of Cachin and Strobl [CS04] works as follows.

- When a server i initializes a session ID, it first chooses a contribution y_i randomly from $\{0,1\}^k$; the goal is to compute the key for this session as $sk = \sum_{j \in G} y_i$ for some set G of $\lceil \frac{n+1}{2} \rceil$ servers. It then generates public key/private encryption keys (PK_i, SK_i), and sends PK_i to every other server.
- When a server S_i receives such a public key PK_j from another server S_j, it sends the contribution value y_i encrypted under PK_j to server S_j. Once it has received the contribution values $y_{u_1}, \ldots, y_{u_{\lceil \frac{n+1}{2} \rceil}}$ of $\lceil \frac{n+1}{2} \rceil$ servers like this, it computes the differences $d_1 \leftarrow y_{u_1} - y_{u_2}, d_2 \leftarrow y_{u_2} - y_{u_3}, \ldots, d_{\lceil \frac{n+1}{2} \rceil} \leftarrow y_{u_{\lceil \frac{n+1}{2} \rceil}} - y_{u_1}$, and proposes the sequences $\langle u_1, \ldots, u_{\lceil \frac{n+1}{2} \rceil} \rangle$ and $\langle d_1, \ldots, d_{\lceil \frac{n+1}{2} \rceil} \rangle$ in the consensus sub-protocol. Note that the difference between any pair of contribution values may be leaked through this, but since no other information is revealed, all contribution values remain secret.
- When a server S_i decides two sequences $\langle \bar{u}_1, \ldots, \bar{u}_{\lceil \frac{n+1}{2} \rceil} \rangle$ and $\langle \bar{d}_1, \ldots, \bar{d}_{\lceil \frac{n+1}{2} \rceil} \rangle$ in the consensus protocol, it computes the session key as follows. It first chooses an arbitrary index $m \in [1, \lceil \frac{n+1}{2} \rceil]$ such that it has received $y_{\bar{u}_m}$ before (notice that such an m exists, as it has received at least $\lceil \frac{n+1}{2} \rceil$ values y_j at this point).

  It then computes the session key $sk = (\sum_{j=1}^{\lceil \frac{n+1}{2} \rceil - 1} j \bar{d}_{m+j}) + (\lceil \frac{n+1}{2} \rceil) y_{\bar{u}_m}$, where $\bar{d}_{m+j} = \bar{d}_{m+j-(\lceil \frac{n+1}{2} \rceil)}$ for $m + j > \lceil \frac{n+1}{2} \rceil$.

It is easy to see that every server terminates. It is also easy to verify that every server computes the same session key $sk = \sum_{j=1}^{\lceil \frac{n+1}{2} \rceil} y_{\bar{u}_j}$, regardless of which m it chooses. Finally, since all contribution values remain secret (as argued above), the same holds for the key sk.

Complexity. The protocol requires every server to perform n encryptions and n decryptions. Furthermore, every server sends $2n$ messages of size k bits in two rounds (k is the size of the public keys used for the encryption scheme), and additionally, requires every server to complete a consensus protocol on a value of size $k \lceil \frac{n+1}{2} \rceil$.

2.4 Implementation for Secure Multicast

Our protocol `SMC` for secure multicasting allows to establish several instances of a secure multicast channel in parallel. Every such instance is identified by a unique tag ID, which may be an arbitrary bit-string. An established instance allows every server to send a message to all other servers, such that nobody outside the group learns the content of the message. Furthermore, it guarantees authenticity of the sender, i.e., if an instance delivers a message m from a server S_j, then S_j has sent m through this instance before.

Specification. More technically, our protocol has the following interface. Every server S_i accepts two types of input messages: (init, ID) and $(\mathsf{multicast}, ID, m)$. If S_i receives an input of the first type, we say S_i *initializes instance ID;* in case of an input of the second type, we say S_i *multicasts m over ID* .

Every server produces also two types of output messages: $(\mathsf{established}, ID)$ and $(\mathsf{received}, ID, m, j)$. If a server S_i produces an output of the first type, we say S_i *establishes instance ID;* if the output is of the second form, we say S_i *delivers m from S_j in ID.* With respect to this interface, our protocol `SMC` satisfies the following properties:

TERMINATION: If at least a majority of servers initialize an instance ID, then every server that initializes ID eventually establishes ID.

Moreover, if a server S_i multicasts m over ID after establishing ID, then every server that established ID at that point eventually delivers m from S_j in ID.

AUTHENTICITY: If a server delivers m from S_j in ID, then S_j has multicast m over ID before.

SECRECY: Only the servers of the multicast group learn the messages being multicast.

EFFICIENCY: Multicasting a message of size l bits over an instance ID requires multicasting $l + k$ bits over the underlying insecure multicast channel, where k is the length of a digital signature under the given public keys.

The Protocol. Our protocol `SMC` for secure multicasting assumes a sub-protocol `GKE` for fault-tolerant GKE, a symmetric encryption scheme, and works as follows.

- When a server S_i initializes an instance ID, it initializes a session $ID|\mathsf{gke}$ of the sub-protocol `GKE`, where gke is an arbitrary public constant.
- When a server S_i computes a key sk in a session $ID|\mathsf{gke}$ of the sub-protocol `GKE`, it stores the key in a local variable sk_{ID}, and outputs $(\mathsf{established}, ID)$.
- When a server S_i multicasts a message m over ID, it waits until it has computed the key sk_{ID}. It then computes a signature σ_i on the message $m||ID||i$ under its secret key s_i, where $||$ denote the concatenation of the values in a well-defined encoding. Next, it computes the encryption c of $m||\sigma_i$ under the key sk_{ID} using the given symmetric encryption scheme, and multicasts (c, ID) over the underlying insecure multicast channel.
- When a server S_i receives a message (c, ID) from a server P_j through the underlying insecure multicast channel, it waits until it has computed the key sk_{ID}, and then decrypts c using sk_{ID} to get m'. Next, it parses m' as $m||\sigma_j$, and verifies the signature σ_j on $m||ID||j$ using the public key p_j. If this verification succeeds, it outputs $(\mathsf{deliver}, ID, m, j)$.

By the termination property of the underlying GKE protocol, it follows immediately that every server that initializes an instance ID eventually establishes ID, provided that a majority (i.e., $\lceil \frac{n+1}{2} \rceil$) of the servers initialize ID. The second part of the termination property of `SMC` then follows immediately by the reliability of the underlying insecure multicast channel.

Authenticity of `SMC` follows by the security of the signature scheme used. We remark that the reason for signing m concatenated with the tag ID is to ensure that messages sent over an instance ID will only be delivered by this instance. Notice, however, that our protocol does not prevent reply attacks, i.e., even though messages are authenticated, they may be delivered multiple times. A higher level application can easily prevent this by adding sequence numbers to the messages it multicasts over an instance, and by accepting a delivered message only once for every sequence number.

Privacy of SMC follows by the secrecy property of the underlying GKE protocol, and the security of the symmetric encryption scheme used. It is also easy to see that efficiency holds: to multicast a message m over an instance ID, a server sends the encryption of $m||\sigma_j$ together with the identifier ID over the underlying channel. These are at most $|m| + |\sigma_j| + |ID|$ bits, assuming that the underling encryption scheme is a one-way permutation, i.e., encryptions of a message have the same length as the message itself (this holds for most symmetric encryption schemes, such as 3DES or AES). As $|m|$ will be typically much bigger than $|ID|$, we may neglect $|ID|$.

Complexity. In order to setup a secure multicast instance ID, our protocol requires one execution of the GKE protocol (see previous section). The overhead for multicasting one message through an instance ID (as opposed to just multicast the message over the underlying insecure multicast channel) consists of computing one digital signature and one symmetric encryption of the message. The overhead for receiving a message consists of one signature verification, and one symmetric decryption.

3 Experimental Results

We now present the experimental costs of our construction for secure multicast. Specifically, we present the costs for setting up a secure multicast channel, and the continuous overhead for authenticating and encrypting multicast messages. The prototype implementation that we used for this measurements is written in Java; we describe its architecture in Section 4.2.

Our testbed for these measurements consists of a cluster of seven 1.2 GHz Pentium III dual-processor computers running Linux 2.4.20. The machines were connected over a 100 bits/s LAN. For the private and public signature key pairs that every server is assume to have in our system model (cf. Section 2.1), we used 512-bit RSA keys.

3.1 Setting up a Multicast Channel

The time needed for setting up a secure multicast channel consists of the time needed for computing a common group key by the GKE protocol described in Section 2.3. Below, we present the performance of our prototype implementation of the GKE protocol. In this implementation, we instantiated the encryption scheme used by the GKE protocol through an RSA encryption scheme. For measuring the running time of the GKE protocol, we started the protocol on all servers simultaneously, and computed the average time it took a server from starting the protocol until it computed the group key. This time involves all the overhead by the underlying consensus, as well as the underlying network.

Our primary goal is to investigate how the running time of the protocol grows in relation to the group size. We therefore measured the running time of the GKE protocol for different

Figure 1: Running time of the GKE protocol

group sizes, ranging from 3 peers up to 21 peers. For all measurements, we used 512-bit RSA encryption keys (within the GKE protocol), and generated a group key of size 128-bits. Figure 3.1 summarizes the results. Notice that the running time grows faster than linear in the number of peers that participate. This growth seems to be induced by the running time of the consensus sub-protocol. The fact that the consensus sub-protocol has a super-linear running time is somewhat surprising, as its theoretical round complexity does not depend on the group size but only on the time needed until the failure detector stabilizes.

We remark that in our measurements, all peers were participating in the protocol. In case that only a majority of peers participates, the protocols may actually run faster, since fewer messages will be sent and the load on the network will be reduced by a factor of two. It would be interesting to investigate this through another series of experiments (due to time constraints, we could not investigate this anymore).

3.2 Continuous Overhead for Securely Multicasting Messages

The continuous overhead for securely multicasting messages consists on the sender-side of one symmetric encryption and one asymmetric signature generation per message. On the receiver-side, it consists of one symmetric decryption and one asymmetric signature verification per message.

For our measurements, we instantiated the symmetric encryption scheme used by our `SMC` protocol by 128-bit AES; for the signature scheme used to authenticate messages, we used the 512-bit RSA signature keys from our testbed. We measured the sender and receiver overhead for different message sizes, ranging from 0 to 1000 kbits. From the time needed for signing and encrypting messages, one can easily derive an upper bound on the "throughput"

Figure 2: The continuous overhead for multicasting messages is shown on the left hand side. The corresponding maximal throughput is shown on the right hand side.

of our secure multicast implementation, i.e., the bits per second that can be transmitted over a secure multicast channel. Figure 3.2 summarizes the overhead, as well as the maximal throughput.

We remark that the actual end-to-end throughput will be lower, as messages may be lost or corrupted when sent over the underlying insecure multicast channel. The rate at which such losses or corruptions occur will depend on the particular network, and will typically increase as the message size increases.

Asymmetric signature generation and verification is typically much slower than symmetric authentication. We therefore also measured an alternative implementation for authenticating multicast messages. In particular, we used a second group key of size 128-bits, and authenticated all messages using HMAC-MD5 under this group key. As a result, the messages are only authentic on a group level and not on a peer level. For certain applications, this may actually be enough. Figure 3.2 summarizes these results. Not surprisingly, this approach outperforms the asymmetric approach by a factor of four.

4 A Peer-to-Peer System for Private Virtual Conferencing

In this section, we describe the architecture and implementation of `SecChat`, which is a peer-to-peer application for private virtual conferencing. `SecChat` allows any peer to create conferences among a subset of the peers, and provides secure multicast communication among the peers of a conference. The secure multicast channel for a conference becomes available as soon as a majority of the designated members have joined the conference.

4.1 Architecture

We consider a system of N peers $P_1, \ldots, P_N$, subsets of which want to repeatedly and concurrently establish a private virtual conference. We assume an X509 public key infrastructure (PKI), which assigns — by means of an X509 certificate — a unique signature public key p_i to every peer P_i; the corresponding private key s_i is only known to the peer P_i.

`SecChat` allows peers to create conferences among each other and to join created conferences. Furthermore, it provides secure multicast communication among peers of a conference. Technically, `SecChat` consists of two components: a directory service, and a protocol for secure multicast. The directory service is used to store conference specific information, and to distribute the necessary public keys and certificates among the conference members. The secure multicast protocol is used to provide secure communication among the members of a conference. On an abstract level, `SecChat` works as follows.

Add Participant: The system allows any peer to add itself as a participant of the system. To do this, a peer has to generate a public and private key pair, and to get an X509 certificate on its public key from the PKI. It then stores the certificate in the directory service.

Get Participant: The system allows each peer to retrieve the public key certificates as well as the names of all participants.

Create: The system allows any peer to create a conference. In order to create a conference, a peer has to provide a unique tag ID for the conference, a time frame t_0, t_1 during which the conference shall be held, as well as the indices (names) G of intended members of the conference together with their public key certificates $C = \{(j, C_j) \mid j \in G\}$. A conference is then created by storing the corresponding tuple (ID, t_0, t_1, G, C) in the directory service.

Join: The system allows any peer to join a conference for which it is a member. In order to join a conference with identifier ID, a peer i first retrieves the corresponding tuple (ID, t_0, t_1, G, C) from the directory service, and verifies all certificates in C. If the verification fails, it aborts the conference. Otherwise, it it starts a secure multicast instance with tag $ID|\mathsf{smc}$ among the peers in G, using the public keys from C.

Send: The system allows a peer that joined a conference with tag ID to send a message m to all other conference members. The message m is sent using the secure multicast instance with tag $ID|\mathsf{smc}$

Receive: If a peer has joined a conference with tag ID, and the secure multicast instance with tag $ID|\mathsf{smc}$ delivers a message m, then the system delivers the message m to the peer for conference ID.

Notice that by assumption, the X509 PKI assigns a unique signature key to every peer, i.e., for every peer, there exists exactly one valid X509 certificate binding a public key to this peer, such that only this peer knows the corresponding private key. This ensures the authenticity of the public keys and thereby the security of the conferences, even in case the directory service behaves maliciously. Specifically, a malicious directory service can only prevent a conference from happening (by providing for example invalid certificates), but not invalidate the authenticity of the public signature keys, as it is impossible (by assumption) to forge certificates.

4.2 A Prototype Implementation in Java

We now proceed with an overview of our prototype implementation of `SecChat`. The prototype is written in Java, and is organized in the following packages:

Figure 3: Architecture of the Prototype Implementation of `SecChat`

`secureMulticast.protocols`: This package contains all classes for implementing the protocols involved in a secure multicast. Specifically, it implements peer-to-peer communication, a failure-detector, a consensus protocol, a GKE protocol, as well as a secure multicast protocol.

`secureMulticast.database`: This package contains all classes for implementing the directory service. We implemented the directory service as a central component running on a server that must be accessible by every peer.

`secureMulticast.gui`: This package contains all classes for the graphical user interface (GUI) of our application.

`secureMulticast.model`: This package contains all classes needed for managing initialization data, such as the private and public key certificates, encryption algorithms used, and so on. It serves as a basis for all other packages.

Figure 4.2 illustrates the dependencies of the packages, and also lists the most important classes of each package. In the following, we describe each of the packages in more detail.

The Protocol Stack (`secureMulticast.protocols`). This package provides a set of protocols used for implementing secure multicast (see Figure 4.2 for a list of all protocols). Every such protocol is a sub-class of the abstract class `Protocol`, the abstract class

`SingletonProtocol`, or of the abstract class `GroupSingletonProtocol`. Protocols of the first type can be instantiated multiple times for the same group, whereas protocols of the second type can only be instantiated once for the same group. Protocols of the third type can only be instantiated once per virtual machine.

For every protocol class *protName*, there is a caller interface `[`*protName*`]Caller` that must be implemented by the object that instantiates the protocol. The interface specifies the methods that will be called by the protocol instance in case it produces an output.

Protocols are instantiated by calling the corresponding static method `Protocol.newInstance` (or `GroupSingletonProtocol.newInstance`, `SingletonProtocol.newInstance`, respectively). This method expects a unique tag *ID*, a reference to the calling object, and an object of type `Group`, which specifies the intended participants. It is then verified if the tag *ID* is unique, and if the calling object implements the corresponding caller interface. In case all checks succeed, the corresponding instance in generated (in case of singleton protocols, it is looked-up in an internal hashtable) and returned to the calling object.

Protocols may be built in a modular way, i.e., they may build on sub-protocols. In the following, we call such a composition of protocols a protocol stack. The intra-stack communication is done by method calls whereas the inter-stack communication is message based. Every message contains the unique protocol *ID* and name of the sender and the receiver. There are two types of messages, peer-messages and group-messages. Regarding which message type the PT2PTLink — a GroupSingletonProtocol — will send the message to all or just to one peer.

For each peer in each group exists a ReceivingThread. This thread waits for incoming messages and puts them in the ReceivingQueue (java 1.4.2 java.nio does not support ssl channels - so we had to give each socket his own listener namely a ReceivingThread). The message will then be handled by a WorkingThread, which delivers the message to the destination protocol and performs all actions started by the protocol upon receiving this message. So the WorkingThread will run through the whole protocol stack if needed.

Messages to other peers may either be send by the WorkingThread in case the message is produced by a protocol or by a thread of the application using this protocol stack.

We now sketch the implementation of the lowest layer of the protocol stack: the protocol `PT2PTLink`. This GroupSingletonProtocol implements the links used by all protocols in order to send messages autheticallly. The authenticity is retrieved by using authentic ssl connections. To verify the authenticity the X509 public key certificates in the SingleKeyStore are used. The SingleKeyStore is a wrapper class for java.security.KeyStore. The SingleKeyStore takes care of loading, storing the KeyStore data and asks the user for the password if needed.

The PT2PTLink starts a ListeningThread to which all other group members may connect. Instantiates the ReceivingQueue and starts as much WorkingThreads as specified in the properties. Upon receiving a connect request the ListeningThread verifies the authenticity,registers the outputstream and address. Creates a ReceivingThread which is then listening on this sslsocket.

Sending a message without having already an established connection will lead to the following. If an address is known for this peer a connection will tried to be opened. If no connection can be established to an other peer, the message is put into a SendingQueue. Upon initialization of the SendingQueue the SendingThread is started which tries to connect and send the messages periodically. The delay between the retries is specified in the properties file. Notice that it is sufficient if one of each pair of peers knows the address of the other to establish a connection. Once a connection was established, both got the address of the other peer, so they can try to reconnect if necessary.

The Directory Service (`secureMulticast.database`). The directory service is build on top of an HSQL-database. In the HSQL-database, java objects can be easily stored as serialized data. The data for this database is stored in the files Test.script, Test.properties. To generate these two files for a new, empty database, run the ant file build.xml. The database is behind a server, which handles the requests. The only purpose of the server is connecting the database to the network. For each request a HandlingThread is started, which takes care of the database updates. The DB-Stub transforms the messages to the corresponding SQL-queries. In case of an add participant it also checks if the certificate is valid. This is necessary, because for each identity (the canonical name of the certificate subject in combination with the certification authority) only one entry may be made in the database. Therefore, an adversary could prevent a peer to participate by adding a forged certificate.

The port to be served by the database server and the path where to find the database files is specified in a property file which path has to be given as argument starting the Server. An example property file can be found under /parameters/DBParameters.secmc.

The User Interface (`secureMulticast.gui`) and Setup (`secureMulticast.model`). A peer is started by invoking the IMFrame with the arguments of your name, and the path to a property file. This property file contains the address of the directory service, the path to the keystorefile, the location of the accepted public certificates, the cryptographic algorithms and providers used, the key sizes, the queue sizes, and the number of WorkingThreads used. Example property files can be found under /parameters/parameters*.secmc.

The communication between the `SecChat` and the directory service is handled by `secureMulticast.protocols.DBConnection`. Joining a conference will firstly perform a check if all certificates of the conference members are already accepted. All accepted certificates are stored in the `secureMulticast.model.SingleKeyStore`. For the not accepted certificates, the user will be asked if he trusts these certificates. After all certificates have been accepted and added to the Singlekey Store, the members of the conference are added to `secureMulticast.model.Peers`. Furthermore a Session object is created and added to `secureMulticast.model.Sessions`. The AuthenticIpGroup (which is needed to start the SMC) is then created from of the Session object. Then, the Sec_Multicast is started to compute the group key, and once the group key is computed, messages will be securely multicasted via Sec_Multicast.send().

A Assessment of personal work

In the beginning, I enjoyed studying the protocols, which I should implement. I spend too much time on finding out which approach to take for the protocol stack. Therefore, I started with the design of `SecChat`. First, the directory service and the model for the chat, then it was time for the GUI. Since this was my first GUI, this took much too long and is still not finished. At that point, I started implementing the protocol stack, thinking that I was already much too late. So I did probably the biggest mistake, I developed the protocol stack without testing. Like that, I could catch up with my time plan. But when I started testing the whole protocol stack, I had to pay for it. Furthermore there have been some problems in setting up a test environment, which cost me quite a lot of time. But trying to set up the test environment, I learned a lot about KNOPPIX (We first wanted to run the tests on Laptops, all exactly the same, running KNOPPIX. But the Laptops were so slow, setting up the Secure Multicast took about 10 times as long as on a 'normal-PIII 1.1GHz' Laptop so we decided to test it on the servers.) The two main conclusions are for me:

- never program faster than you can test
- never work too much, because you will have to pay for your overhours with inproductivity tomorrow

References

[Asp02] James Aspnes. Randomized protocols for asynchronous consensus. Manuscript, available from `http://arxiv.org/cs/0209014`, 2002.

[BCK98] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, 1998.

[BPR00] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology: Eurocrypt '00*, 2000.

[CGI$^+$99] Ran Canetti, Juan Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and efficient constructions. In *Proc. INFOCOM '99*, 1999.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 42–51, 1993. Updated version available from `http://www.research.ibm.com/security/`.

[CS04] Christian Cachin and Reto Strobl. Asynchronous group key exchange with failures. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Rei82] R. Reischuk. A new solution for the byzantine general's problem. Computer Science Technical Report RJ 3673, IBM Research Laboratory, 1982.

[Sho99] Victor Shoup. On formal models for secure key exchange. Research Report RZ 3120, IBM Research, 1999.