UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ATTACK-TOLERANT COMMUNICATION IN A SIEM TOOL

Ricardo Jorge Pato Fonseca

PROJECTO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2012

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

ATTACK-TOLERANT COMMUNICATION IN A SIEM TOOL

Ricardo Jorge Pato Fonseca

PROJECTO

Projecto orientado pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2012

Acknowledgments

First of all, I would like to thank my supervisor, Professor Nuno Ferreira Neves, for introducing me to the project MASSIF and giving me the opportunity to work in the project and write a thesis on the subject. I am also particularly grateful for the dedicated support and advice he has given me throughout the year, for without it this thesis could not have been completed.

I would also like to thank the people associated with the MASSIF project, including members of the Navigators team, who, besides other tasks related to the MASSIF project, also discussed ideas on the REB over the several meetings that occurred throughout the year. These people include Professor Alysson Bessani, Professor António Casimiro, Professor Paulo Veríssimo, Eric Vial, Hossein Rouhani and Miguel Garcia.

Finally, I would like to thank my family who have given me unconditional support, especially in the final stages of the thesis.

Lisboa, September 26, 2012
Ricardo Fonseca

Resumo

Uma ferramenta SIEM (*Security Information and Event Management*) representa uma solução que combina a coleção de eventos em tempo real através da monitorização de uma rede de computadores em múltiplas localizações, com a correlação e análise dos dados obtidos pela monitorização, a fim de descobrir se existem ataques/intrusões em progresso para que alarmes e ações de remediação possam ser iniciadas.

”Prevenção é ideal, mas deteção é obrigatória”. Este é o lema utilizado pelo projeto europeu MASSIF (FP7-257475) [1] que visa utilizar tecnologias SIEM para oferecer deteção de ataques e intrusões em sistemas informáticos sob variados contextos. Hoje em dia, a tecnologia SIEM não é novidade, no entanto, o projeto MASSIF aborda diversas limitações existentes nos sistemas SIEM atuais, e tenciona apresentar soluções inovadoras para esses problemas. Uma das limitações presentes é a incapacidade do sistema de fornecer um elevado grau de confiabilidade ou resiliência a ataques na disseminação de eventos entre os seus componentes (sensores e motor de correlação). O projeto MASSIF tem como objetivo contribuir para a resolução desse problema, fornecendo um mecanismo de comunicação que assegure a entrega fiável e atempada dos eventos de segurança, ainda que este se encontre sob ataque, quer nos nós que o concretizam como na rede que os liga. Esta tese foi desenvolvida neste contexto, oferecendo uma solução para a comunicação confiável em SIEMs.

Uma vez que sensores e motores de correlação podem encontrar-se geograficamente dispersos, existe a necessidade de fornecer um sistema de comunicação capaz de transmitir dados (e.g., eventos e comandos de reconfiguração) não só em redes de estrutura LAN, mas também em redes de larga escala como a Internet. Alguns dos problemas associados a esta comunicação são atrasos nas transmissões de dados, configurações incorretas de rotas, violações de integridade nos dados transmitidos, e ataques de negação de serviço (DoS). Todos estes problemas podem potencialmente afetar a conformidade da análise de eventos. Na tese é apresentada a conceção de um *Resilient Event Bus* (REB) para uma comunicação confiável entre sensores e motores de correlação. O REB é baseado numa rede sobreposta que opera sobre a infraestrutura SIEM existente, e que usa vários mecanismos, como algoritmos de codificação, *multihoming* e transmissão de dados por múltiplas rotas, para assegurar uma entrega atempada de dados em vários cenários de falha (i.e., do tipo acidental ou malicioso).

A tese apresenta o desenho dos vários mecanismos empregues pelo REB, incluindo uma comparação com outras soluções na literatura atual e a justificação das escolhas feitas. O trabalho também inclui uma descrição de uma primeira concretização do REB, a metodologia utilizada na sua execução e a análise de alguns resultados de testes.

Palavras-chave: Segurança, SIEM, Comunicação tolerante a ataques, MASSIF, REB

Abstract

A *Security Information and Event Management* (SIEM) tool is a security solution that combines real-time event collection by monitoring a target network at a diverse set of locations, with the correlation and analysis of the obtained data to find out if attacks/intrusions are in progress, so that alarms and remediation actions can be initiated. Since the different components of a SIEM tool (e.g., the sensors and the correlation engine) may be geographically dispersed, there is the need for a communication subsystem capable of transmitting data (e.g., events and reconfiguration commands) not only in LAN settings, but also over large scale networks such as the Internet. Some of the inherent problems associated to this communication are delays, routing misconfigurations, message integrity violations and denial of service attacks, which can all affect the correctness of the event analysis. This thesis presents the design of a Resilient Event Bus (REB) for a resilient communication across a diverse set of network environments. The REB is based on an overlay network superimposed on top of the existing SIEM infrastructure, which uses several mechanisms, such as coding algorithms, multihoming and multipath data transmission, to ensure timely data delivery in various failure scenarios (i.e., of both accidental and malicious nature).

Keywords: Security, SIEM, Attack tolerant communication, MASSIF, REB

Contents

List of Figures

Chapter 1

Introduction

Concerning Information Security, multiple strategies exist to defend both services and critical data from non-authorized access or external attacks. One particular strategy is *prevention*, which attempts to ensure the confidentiality and integrity of data, authentication of involved entities, and availability of services. However, despite prevention being an interesting strategy, there is the issue of new forms of attacks being found frequently, which makes imperative the usage of detection mechanisms.

Attack and intrusion detection in computer systems assumes a constant monitoring of those systems, as well as a simultaneous analysis of the monitoring information records. Typically, the monitoring consists in generating *log* files of events that contain specific information about particular services, which are delivered afterwards to an entity capable of analysing those events and extracting relevant information from the data (such as data patterns which indicate that "something is not right"). That information can then be used afterwards to correct the perceived anomaly or to present evidence in court against the attacking entity, if one exists.

An inherent problem in the detection strategy is the effort usually required to extract the information from the monitoring events. This task typically involves filtering relevant events, translating those events into contextual information, and sometimes making that information available for auditing purposes. When an organization chooses to apply this security strategy, the human effort typically increases, thus becoming advantageous to the industry the development of solutions that automate (to a certain extent) the management of events and security information.

The concept of *Security Information and Event Management* (SIEM) was first publicly mentioned in 2005 by Mark Nicolett and Amrit Williams, with the intent of unifying the two existing concepts: *SEM* (events), which relates to the real-time analysis of events obtained from the monitoring of systems; and *SIM* (information), which relates to the long-term storage of large amounts of information extracted from the analysis of events, and later historical analysis of the same information in order to support forensic activities [25]. A SIEM tool offers multiple advantages over the human approach, allowing

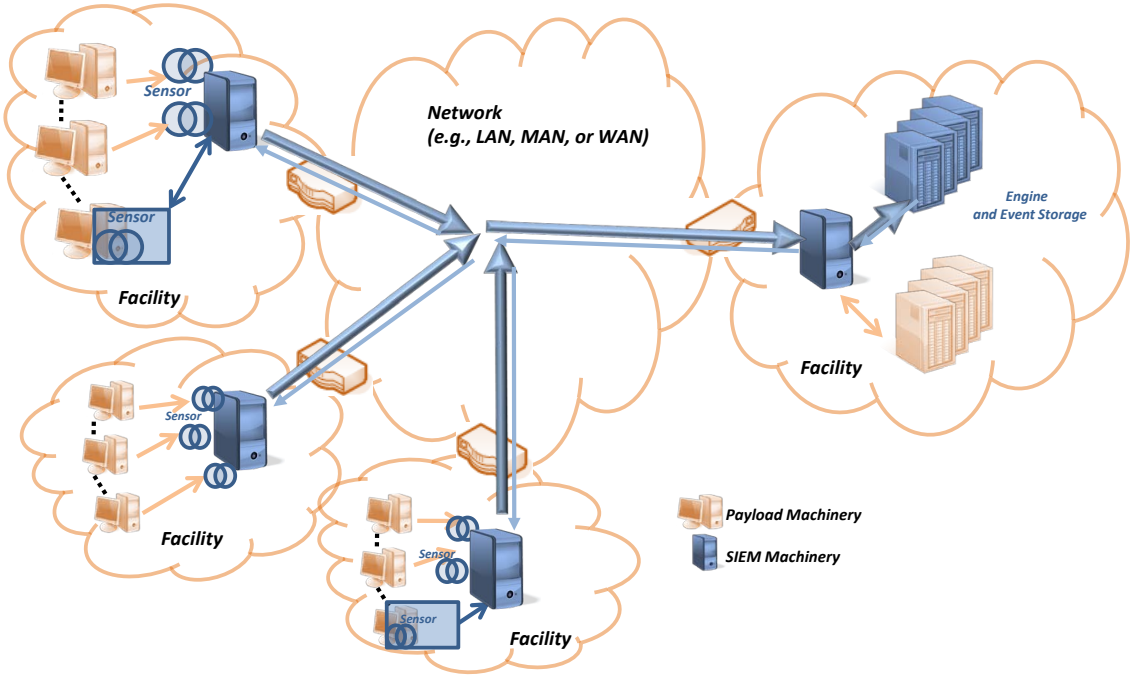

Figure 1.1: SIEM general architecture.

a faster reaction to an attack, improving the efficiency of attack/intrusion detection, and automating the process of information compliance for auditing purposes [10, 11, 18]. Currently, they are being employed by organizations around the world as a way to facilitate operations related to maintenance, monitoring and analysis of networks, by integrating a large range of security and network capabilities, which allow for instance the correlation of thousands of events and the reporting of attacks and intrusions in near real-time.

There are two main components in a SIEM tool: sensors and a correlation engine (see Figure 1.1). Sensors are responsible for periodical event generation based on real-time monitoring of selected network components (e.g., routers, firewalls, intrusion detection systems). These components are called the payload machinery, and are available on the existing network infrastructure (left part of the figure). The correlation engine is responsible for event data analysis and for extracting useful information that could aid in the detection and remediation of unusual situations, such as system intrusions (right part of the figure).

1.1 Context

The work of this thesis was developed in the context of the European project MASSIF [1] where the objective is to achieve significant advances in the area of SIEM system development and deployment. Deliverable document D5.1.1 [16] describes a preliminary architecture of a *resilient* SIEM tool for the project, including a short description of the middleware services that could be employed to build the components of the SIEM. Out of

those services, there is a communication subsystem between the sensors and the engine in a SIEM system, which is the theme of this thesis.

1.2 The need for resilient event dissemination

Due to its fundamental role in the security management of an organization, the SIEM should be made resilient to faults and attacks, and one prominent attack surface is the communication between sensors and the correlation engine. This communication primarily consists in the transmission of events from the sensors to the engine, but sometimes it can include also remediation actions that are sent towards the payload machinery, for instance to reconfigure a certain device after the detection of an intrusion. Depending on the organizations, sensors and the engine may be located near to each other (e.g., through a LAN) or be geographically dispersed, requiring the exchange of data across a large scale network such as the Internet (e.g., if the SIEM collects data from the offices of an organization located in different cities or even countries). Some of the inherent problems associated to this communication are accidental faults and attacks, which can for instance delay, remove, or corrupt individual packets.

1.3 Overview of the Resilient Event Bus

To counteract the problem in the communication between sensors and the correlation engine we propose a solution that provides both resilient and timely communication, called the *Resilient Event Bus* (or REB). Figure 1.2 illustrates the REB applied to the communication in a SIEM. Using REB, a SIEM machine (sensor or engine) can transmit messages to other SIEM machines in a way that tolerates (accidental or malicious) faults in the network and in some cases faults on REB itself.

The REB consists of an overlay network built among the SIEM nodes, which are called generically as *REB nodes*. The communication in the overlay network is performed on top of the UDP/IP protocols, allowing the support of different network settings. REB uses application-level one-hop source routing to send messages towards the destination, instead of simply following the routes imposed by the network-level routing. It also takes advantage of coding techniques and the available redundancy of the network, such as when a node has multiple network connections (i.e., *multihoming*), to ensure that messages arrive securely and timely with a very high probability.

The overlay network created among the REB nodes allows for multiple distinct routes (or paths) to be taken to transmit data to a specific destination. The REB uses *multipath communication* to send data concurrently over several different paths in the overlay network in order to tolerate problems in the underlay network, such as delays or routing misconfigurations, which may have accidental causes or may be triggered by denial of

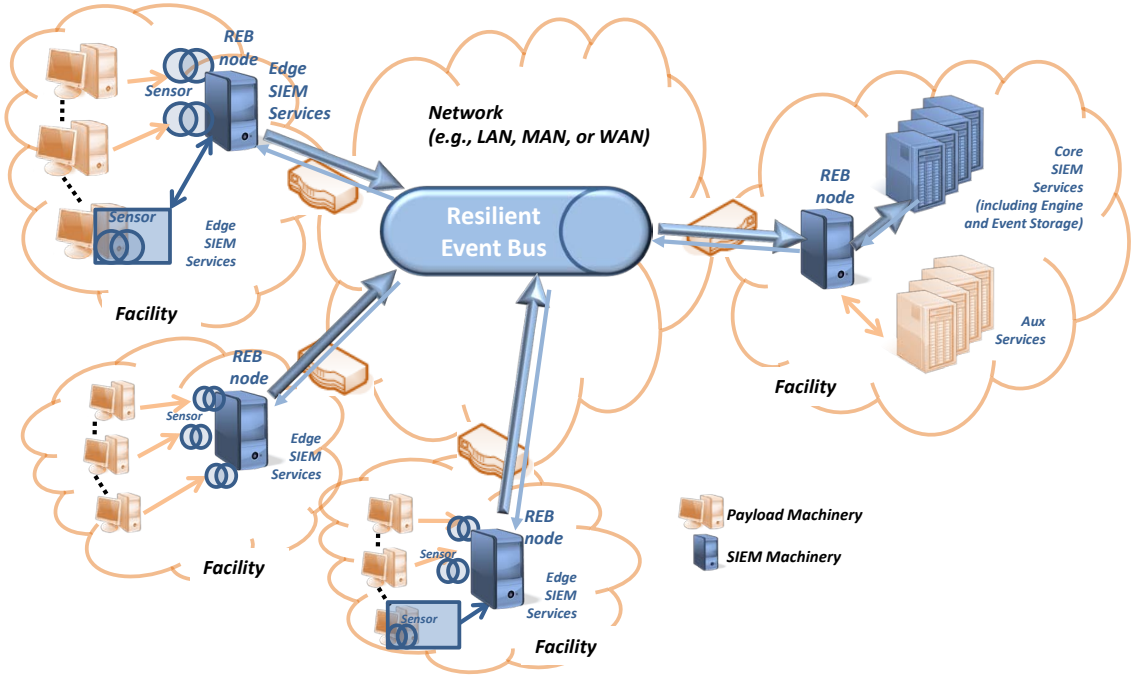

Figure 1.2: The *Resilient Event Bus* applied to the communication in a SIEM.

service attacks. The overlay routes consist either of direct paths between the source and destination nodes, or paths in which an intermediary node receives data from a source and redirects it to the destination. The overlay route of each message is defined at the sender (source routing), based on the local knowledge of the state of the links, and is composed of at most one intermediate relaying REB node (one-hop). Using multiple paths, the communication also tolerates accidental faults in some intermediary nodes.

A REB node can be connected to the WAN through one physical link or through multiple links. The REB takes advantage of this form of redundancy by employing *multihoming communication*. Here, a node can either send or receive packets over any of the physical connections, thus increasing the number of possible paths that can be chosen when disseminating data to a certain destination. For example, if the sender and the receiver have two connections, then there are four alternative direct links between the two nodes instead of a single one. It is expected that at least in part these links fail in an independent way, which means they can be used to provide correct packet delivery under somewhat adverse network conditions.

To avoid replicating messages through multiple links, the REB employs *erasure coding techniques* to ensure that packet loss can be recovered at the receiver, at a much lower replication overhead. Basically, the sender needs to do some processing on a message before transmitting it, producing some extra repair information. Then, a slightly larger message containing the original message and the repair information is divided in several packets that are transmitted over various links. Even if only a subset of the message and repair data arrives, the receiver can still recover the original message.

Communication within the REB nodes uses cryptographic *Message Authentication Codes* (MACs) [24] in order to provide authentication and message integrity. A MAC is appended to each packet a source node transmits to a destination, and is generated using one secret symmetric key associated with the communication session between the source and destination nodes. This form of authentication also makes it possible for the communication to tolerate malicious faults in intermediary nodes.

In order to achieve timeliness with high probability, a sender REB node periodically *probes* each of its most promising paths to a given destination to derive a quality metric to be associated with the path. Based on this metric, the sender can determine at each moment which are the best paths for a destination, and select them to disseminate a message based on its deadline.

1.4 Planning

The thesis work started on the 1st of September 2011, and finished on September 26, 2012. Originally, the project was expected to end in July 2012, however some extra time was necessary to complete the solution. After a first design and implementation of the REB, the analysis of the performance results showed some limitations in certain scenarios that have to be considered in MASSIF. In order to solve these limitations, it was necessary to re-write some of the mechanisms and introduce a few others. In the end, this contributed to an overall better solution.

1.5 The structure of the document

This document is organized as follows:

- In chapter 2 we mention some related work in the area of Overlay Networks and Erasure Codes, and compare them with the REB;
- In chapter 3 we discuss the analysis and design of the REB, explaining its properties and communication mechanisms in detail;
- In chapter 4 we present the implementation and evaluation of our prototype, giving a detailed explanation of its components, referring the utilized methodology, and showing some test results;
- Finally, in chapter 5 we present our conclusions and proposals for future work.

Chapter 2

Related work

Related work to the REB can be divided in two categories: overlay networks and erasure codes. Extensive work has been done over the past years in the area of overlay networks and their benefits. Erasure codes, on the other hand, despite not being new, only more recently become increasingly referenced in various works with practical communication solutions.

2.1 Overlay networks

Andersen et al. talked about how to improve the routing protocols of WANs (like the Internet) using a *Resilient Overlay Network* (or RON) [4]. Such a routing protocol, like the Border Gateway Protocol (BGP), is used in communications among different Autonomous Systems (ASes) which comprise the global structure of the Internet. BGP carries very simplistic routing information for scalability reasons and for policy enforcement since ASes maintain detailed routing information of the subsystems within themselves, not sharing them with other ASes. For this reason, BGP usually takes minutes to recover from faults occurring on the announced routes. This can have a severe impact on the performance of communication between hosts on different ASes, or even affect the overall availability of the communication if it is the case that a particular router is attacked. To overcome these problems, a RON network is used by deploying the RON nodes on distinct ASes and allowing an application-level routing to take place. RON nodes gain knowledge about particular overlay paths by constantly probing those paths, checking their metrics of quality of service such as latency or loss rate, and keeping updated routing tables that contain the "best available route". The best route between a source and a destination RON node may be simply the direct underlay path between them, or it may be a route that uses an intermediary node to forward the data. Either way, cooperation between the nodes is required to keep the routing information updated (the nodes periodically disseminate this information among themselves).

Some design choices of REB build upon the ideas of RON, that is, REB uses overlay

routing to tolerate faults in the underlay network and uses a probing mechanism to select the best route(s). However, in the case of REB, the nodes cannot trust each other to exchange routing information, and so REB uses a source-based routing mechanism where the information of a route is only managed locally by a REB node.

Snoeren et al. discuss the use of a mesh network that is used to disseminate XML packets [23]. Here, the packets are transmitted over multiple disjoint paths, as a form of redundancy, in order to increase the chances of delivery in case faults occur in the underlay network. In this work, the authors sacrifice bandwidth for reliability.

REB uses multipath transmission but does not simply duplicate the transmission of packets over multiple routes, instead it relies on erasure codes to minimize the message overhead caused by the redundancy (necessary for achieving robustness in the communication). Furthermore, REB attempts to combine a best-route strategy from RON with a multi route strategy from mesh networks, compromising on the number of selected routes for transmission of data, that is, only using a subset of all the available routes, but always picking the best routes through continuous probing.

Guo et al. [9] and Akella et al. [2] individually talk about the advantages of multihoming in communication (whether an overlay network is being used or not), stressing the point that multihoming provides minimum path correlation, which is advantageous in an overlay scenario because it supports the idea of independent path failures. This assumption was partially taken by RON, however, RON suggested that overlay nodes should be deployed over distinct ASes in order to better tolerate faults within one particular AS.

REB uses multihoming, where an overlay node utilizes multiple network addresses, each one ideally connected to a different ISP, to increase the chances that any two different overlay paths are disjoint (and tolerate single path failure).

Obelheiro et al. propose an extension to RON where authentication and message integrity are added to the communication [19]. Their proposal is called a *Lightweight Intrusion-Tolerant Overlay Network* (or LITON) which ensures connectivity even in the presence of faults and intrusions in the (overlay and underlay) networks. LITON uses an on-demand route request mechanism that serves a better purpose in MANETs where bandwidth usage should be minimized. In order to tolerate attacks in the transmission of route replies, LITON applies a chain of MACs to a reply, one MAC for each node of the overlay route. This mechanism, however, adds a significant overhead in the establishment (or update) of a route.

REB also uses MACs to provide authentication and message integrity in the communication. However, following the conclusion of Gummadi et al. [8], REB only uses at most one-hop source routing (therefore only using at most two MACs in a single packet), since

more than one-hop does not improve significantly the fault tolerance aspect of overlay networks when these are used to provide better connectivity to the main communication. Unlike LITON, REB does not use an on-demand route request mechanism, since all REB nodes contain the knowledge of the whole network.

2.2 Erasure codes

Error-correcting codes are used to tolerate losses in a communication channel. These codes apply an error correction mechanism to reconstruct the original data from a corrupted packet (i.e., individual bits within the packet are "flipped" from their original value). The trade-off is an increase in the overhead where additional repair data must be added to individual packets in order for the mechanism to succeed. An erasure code is a specific type of error-correcting code, which assumes a binary erasure channel – a channel where a packet either arrives "intact", or does not arrive at all (packet corruptions are viewed as packets not being received). Erasure codes tolerate data loss by encoding information from certain packets into others, so that the loss of a subset of packets does not preclude the possibility of data reconstruction at the receiver.

Since REB uses MACs to provide, for instance, message integrity, individual packet corruptions can be detected and those packets are simply discarded. This makes the usage of erasure codes ideal in this scenario because the assumption of a binary erasure channel can be taken.

Erasure codes have the advantage of being more efficient on larger size data and requiring less total overhead in the communication when compared to other error correcting codes (instead of a fixed amount of extra bits per packet, a fixed amount of extra *packets* is transmitted). One such code is the LT code [12], designed by Michael Luby.

LT codes are a rateless kind of erasure codes, that is, they divide a segment of data in multiple blocks and produce an infinite number of encoded blocks that contain data from one or more original blocks. Furthermore, LT codes guarantee that the original data can be recovered with very high probability if the number of received encoded blocks is slightly larger than the number of original blocks. They use a specially constructed distribution to choose which original blocks to encode in a particular encoded block, the Robust Soliton Distribution.

REB utilizes LT codes combined with a multipath strategy in order to avoid simply replicating data over multiple paths, instead sending individual encoded blocks through different routes. This has the benefit that even if one or more paths is behaving in an incorrect way, the remaining blocks transmitted through the "good" routes will typically be sufficient to overcome the loss rate in the network.

Codes of the kind of the LT code are called Fountain Codes because they provide a continuous flow of data much like a fountain produces a continuous flow of liquid [14].

These fountain codes have successful applications in data dissemination since they provide an efficient and scalable solution to content delivery to a large number of destinations.

One of the early proposed applications of erasure codes is presented by Byers et al. [6]. The authors propose a mechanism for content delivery on adaptive overlay networks, using fountain codes to disseminate the data. However, since destination overlay nodes also retransmit the received encoded blocks in order to perpetuate the bulk transfer, care must be taken to avoid significant levels of transmission redundancy, otherwise the effective solution of fountain codes loses its properties. The authors address this issue by making the nodes cooperate among themselves using reconciliation strategies with different granularities, such as working set sketches (coarse-grained) and bloom filters (fine-grained), in order to limit redundant retransmissions, that is, retransmissions of the same exact blocks that occur on different nodes.

Fountain codes have also evolved over the years in terms of efficiency and the codes with highest performance nowadays are the *Raptor codes*, invented by Shokrollahi [22]. Unlike LT codes, which show different time complexities in the encoding and decoding processes (the encoding showing linear time and the decoding showing worse than linear), Raptor codes perform in linear time on both encoding and decoding processes. The latest generation of Raptor codes is called RaptorQ [13].

Chapter 3

REB Analysis and Design

This chapter presents in detail the design of the Resilient Event Bus (REB).

The REB is organized as an overlay network built among some of the SIEM nodes. The communication in the overlay network is performed on top of the UDP/IP protocols, allowing the support of different network settings. REB uses application-level one-hop source routing to send messages towards the destination, instead of simply following the routes imposed by the network-level routing. It also takes advantage of coding techniques and the available redundancy of the network, such as when a node has multiple network connections (i.e., multihoming), to ensure that messages arrive securely and timely with a very high probability.

3.1 Overview

The REB design is influenced by the way SIEM systems are deployed, which are usually distributed over several facilities. A facility corresponds to a subset of the overall network, where either a set of sensors collect event data or where a group of machines implement the engine and other supporting services. In MASSIF, there is a special device placed in every facility, which is called a MIS. Near the edge of the network, this device aggregates the events obtained locally by the sensors, and forwards them towards the core facility where the engine is located (see left side of Figure 3.1, where these MIS are named edge-MIS). The MIS in the core facility receives the event data from all other MIS, and then re-transmits it to its final destination, which is usually the correlation engine (see the right side of Figure 3.1, where this MIS is called core-MIS). The REB is executed by these MIS devices, and therefore, each one of them has a REB node.

A facility can therefore be modeled as a LAN, and the associated MIS can be seen as a routing device that receives the data produced locally and forwards it towards the final destination facility. The interconnection among the facilities can be abstracted as a WAN. One however should keep in mind that LAN and WAN are modeling artifacts, since in practice they will depend on the actual deployment of the SIEM. In one extreme

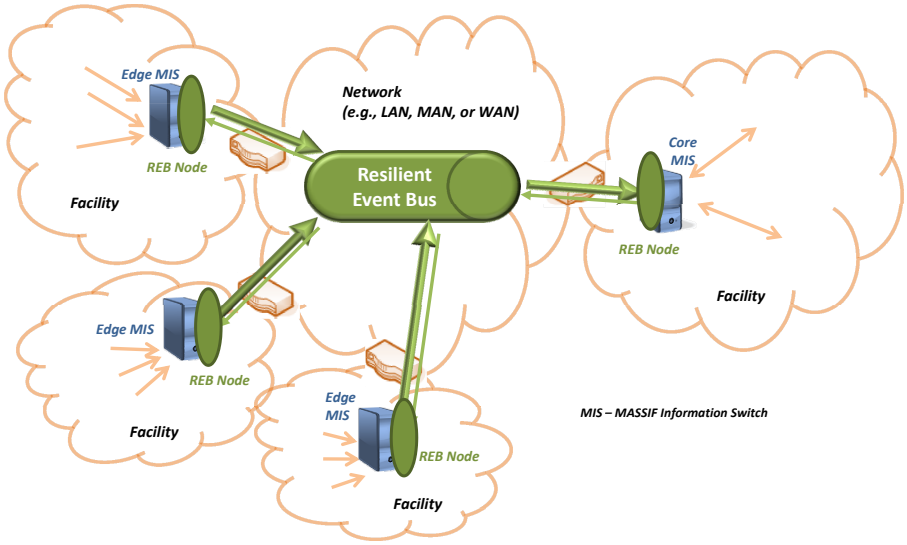

Figure 3.1: REB topological view.

case, the WAN can be the Internet, if the SIEM collects information from various offices of an organization that are located in different regions (of the same country or different countries). In the other extreme case, the WAN could be a set of switches with virtual LANs that interconnect a few PC racks on a data center. This organization has the virtue that entails no meaningful modification to the existing SIEM system and only requires the introduction of a REB node in each MIS at the border of a facility.

The communication among REB nodes is through the UDP/IP protocols. REB nodes can define an overlay network atop the IP network, and run application-level routing strategies to select overlay channels that are (expectedly) providing correct communication. Overlay networks have been used as mechanisms to implement routing schemes that take into account specific application requirements [4]. In MASSIF we want to employ overlay networks to create redundant network-agnostic channels for robust and timely communication, namely for event transport from the edge sensors to the core event correlation engine.

Depending on the network setting, we envision different kinds of faults that might preclude the transmission of data unless appropriate measures are enforced among the REB nodes. For instance, accidental or/and malicious faults can cause the corruption, loss, re-order and delay of packets. An adversary might try to modify the event data carried in a packet or add new events to prevent the detection of an intrusion. Congestion or denial of service (DOS) attacks can also make certain IP routers between specific REB nodes unresponsive, causing significant packet loss or the postponement of their delivery. A strong adversary might also be able to take control of one of the IP routers, allowing her the capability to perform sophisticated forms of man-in-the-middle attacks. REB nodes may also suffer failures, such as a crash or a compromise by an adversary. In this case,

the REB per se will not solve the specific problem of the failed node, and consequently a facility might become disconnected. However, as a whole, the REB should continue to work properly allowing the remaining nodes to continue to exchange data.

Many of the above threats can be addressed with well known security mechanisms that have been applied to standard communication protocols, such as SSL/TLS [7] and IPsec [20]. For example, each pair of REB nodes shares a symmetric secret key. Leveraging from this key, one can add to every packet a Message Authentication Code (MAC) [24] that allows the integrity and authenticity of the arriving packets to be checked in an efficient way. Consequently, any packet that deviates from the expected, either because it was modified or was transmitted from a malicious source, can be identified and deleted. This means that insertion attacks are simply avoided, and corruptions are transformed into erasures that can be recovered with retransmissions (or in our case, also with coding techniques). Optionally, one can also encrypt the packet contents, which guarantees confidentiality and prevents eavesdropping of event data.

Based on these mechanisms, the REB is able to reduce substantially the attack surface that might be exploited by an adversary. However, they only put the REB at the same level of resilience as standard solutions, which are insufficient to prevent certain (sometimes more advanced) attacks. For example, an adversary might thwart the correlation of two sources of events, if she is able to delay the packets that contain the events from one of the sources. This occurs because events are correlated within a predefined time window, and if they arrive in different windows the associated rule might not be activated. Alternatively, the adversary might perform a DOS in one of the IP routers that forwards the packets from one of the sources, causing very high transmission losses and the continuous retransmission of packets (to recover from the packet drops). The overall effect of the attack is once again the delay of the certain event packets, or eventually the temporary disconnection of one of the sources. Traditional solutions for secure communication, such as those based on SSL over TCP, are also vulnerable to other more specific attacks [5]. For instance, this approach is fragile to an adversary with access to the channel between the two communicating parties, since she can continuously abort the establishment of any TCP connection by sending Reset packets, creating in fact unavailability on the communications.

This sort of problems are addressed by the REB by employing some techniques that exploit distinct forms of redundancy. In particular, spatial and temporal redundancy is explored to attain high levels of robustness and timeliness.

Robustness The overlay network created among the REB nodes allows for multiple distinct routes (or paths) to be taken to transmit data to a specific destination. The source can, for instance, send the data directly or ask one of the other REB nodes to forward it to the destination. It is expected, in particular in large scale SIEM deployments, that these two paths will go through distinct physical links, and there-

fore, localized failures will only disrupt part of the communication. Based on this insight, the REB uses *multipath communication* to send data concurrently over several different paths in the overlay network. These routes consist either of direct paths between the source and destination nodes, or paths in which an intermediary node receives data from a source and redirects it to the destination. REB resorts to a one-hop source routing scheme. The overlay route of each message is defined at the sender (source routing), based on the local knowledge of the state of the links, and is composed of at most one intermediate relaying REB node (one-hop). The option of having a single hop, i.e., a single intermediate node, is due to the conclusion of Gummadi et al. [8] that there is no considerable benefit in using more hops.

A REB node can be connected to the WAN through one physical link or through multiple links. The later case is often observed in organizations that operate in critical sectors, such as in electrical production and distribution, as a way to increase their resilience to accidental failures by ensuring that the links are physically separated (they contract the network service to multiple ISPs, and in some cases they go to the extent of using diverse network technologies, one wired and another wireless). If available, the REB also takes advantage of this form of redundancy by employing *multihoming communication*. Here, a node can either send or receive packets over any of the physical connections, thus increasing the number of possible paths that can be chosen when disseminating data to a certain destination. For example, if the sender and the receiver have two connections, then there are four alternative direct links between the two nodes instead of a single one. It is expected that at least in part these links fail in an independent way, which means they can be used to provide correct packet delivery under somewhat adverse network conditions.

Given a certain application message m, the REB could send a copy of it over k distinct paths. This would allow $k - 1$ path failures to be tolerated, at the cost of the transmission of $k - 1$ extra message replicas — the overhead is $(k - 1) * m$. Depending on the message size, this cost can be significant especially when the network is behaving correctly (which is the expected normal case). Additionally, it also requires the receiver to process and discard $k - 1$ duplicates, which can be a limitation in a SIEM system given the asymmetric data flow (remember that the core-MIS needs to receive all traffic coming from the edges). To address this difficulty, the REB employs *erasure coding techniques* to ensure that packet loss can be recovered at the receiver, at a much lower replication overhead. Basically, the sender needs to do some processing on the message m to produce some extra repair information r, and then $m+r$ is divided in several packets that are transmitted over various links. Even if only a subset of the message and repair data arrives, the receiver can still recover the original message.

Timeliness Messages should be transmitted respecting some delivery deadline. The objective is to make the events be processed at the correlation engine while they are (temporally) valid, which requires the REB to enforce timeliness properties of the communication. One should thus assume that there is eventual synchrony, that is, assume that message transmission latency is bounded. However, we must note that the underlying infrastructure can be the target of performance instability, or of attacks (is not trusted by default) which impact on the coverage of those latency assumptions. Although it may be difficult to state the exact bound, specific bounds have to be assumed at run-time, which means that the network will alternate between synchronous and asynchronous behavior, which is undesirable for our objective. As we have seen, the overlay network can provide the necessary path redundancy to provide for timing fault-tolerance. Unfortunately, all overlay networks proposed in the past did not have this objective[1], and therefore, a specific solution had to be developed for the REB.

The REB uses fundamentally two mechanisms to achieve timeliness with high probability, despite being built on top of a best effort time-agnostic protocol such as IP. A sender REB node periodically *probes* each of its most promising paths to a given destination to derive a quality metric to be associated with the path. This metric is currently calculated based on the estimated latency and loss rate, but in the future we are considering other criteria such as the level of independence of this path in relation to the other ones (i.e., to what extent they share the same IP routers). Based on this metric, the sender can determine at each moment which are the best paths for a destination, and select them to disseminate a message based on its deadline.

On average, the above mechanism is able to address most timeliness problems. However, since the metric is calculated based on actual data collected from the network, it may require some period of time for the value of the metric to adjust after sudden changes in the network. During this period the sender could still think that a specific path is good, and consequently continue to use the path for transmissions, when in fact most packets could be lost. Moreover, a malicious REB node could attack the measurement process, for instance by making its paths look particularly good, and then suddenly start dropping all packets. In the REB, these failures end up being tolerated automatically by the erasure codes together with a sufficiently high level of multipath communication. The codes are able to recover from a reasonable number of arbitrary packet losses (in the original and/or repair data). Therefore, if one assumes that the failure only affects a limited number of channels, the remaining ones still deliver enough packets for the original message to be reconstructed.

[1] In the past, some approaches had the aim of improving the end-to-end communication latency, but not of attaining application-defined maximum delays (e.g., [3, 23]).

3.2 Communication properties

SIEM systems are often built directly on top of the TCP protocol, or resort to SSL/TLS to augment TCP with security capabilities. Consequently, SIEM developers expect a communication substrate that provides a set of properties that matches those of TCP, since they greatly simplify the implementation of the system. For this reason, in addition to robustness and timeliness, REB was designed to grant most of TCP properties.

REB exports a relatively simple interface, offering point-to-point communication channels between nodes. Data is transmitted as a stream of bytes, and is delivered reliably in First In First Out (FIFO) order. The arrival of duplicate data is identified and removed, and flow control is enforced at the senders to prevent receivers from being overwhelmed with too much information. REB also ensures data integrity and authenticity, something that TCP does not provide by itself, but that can be attained, for instance, with SSL/TLS. Confidentiality can also be guaranteed on an optional basis, depending on an indication by the applications.

Since the REB is implemented on top of UDP/IP, which does not have any of these properties, it is necessary to devise ways to implement them with a group of mechanisms. In the rest of this section, we explain generically how these properties are attained.

3.2.1 Authentication and error-free (and optional confidentiality)

SIEM applications call the REB interface to send messages to a certain destination. REB treats these messages as sets of bytes that are stored in a queue for transmission. Depending on the amount of queued data and on the maximum transmission unit (MTU) of the underlying network, it might be necessary to break the data first in segments and next in several packets that are forwarded independently (see Section 3.3). REB nodes disseminate the packets using multiple concurrent routes, which can be based on a single direct channel between the source and the destination, or can have a channel from the source to an intermediary node and then another channel to the final receiver. Assuming that the sender and the receiver are both correct, then attacks can occur both on the network and on the intermediary node (in case this node was compromised).

The initialization of the REB creates two shared cryptographic keys between every pair of nodes. The keys are used to protect the communications from attacks, supporting the authentication of nodes and the integrity/authenticity of the data. One of the keys is used to generate a MAC that is appended to every packet. MACs are verified at the receiver before packet delivery, and packets are discarded if their MACs do not match the expected values.

In alternative, one could choose to append a MAC to the whole segment and verify it only after the full reception. This solution would have the virtue of saving some MAC calculations, both at the sender and receiver, whenever segments are large. However, it

suffers from a few drawbacks, making it less appealing in practice. First, the verification would be postponed, allowing corrupted packets to occupy space on the receiver buffers until much later. Second, the receiver node cannot separate good from bad packets, and therefore, a single damaged packet would cause the whole segment to be dropped (and then later retransmitted again).

On direct channels, only one MAC is generated per packet by the source, using the key shared with the destination node. On two-hop channels, a pair MACs is created, the first for the intermediary node and the second for the destination node. After receiving a packet, the intermediary validates and removes the second MAC and only then it forwards the packet to the destination.

Here, again, one could save a few MAC calculations if a single MAC was added independently of the type of route (i.e., direct or two-hop). However, since MACs are obtained relatively fast, we decided that it was better to have the capability to immediately identify and delete modified packets, both at the intermediary and final nodes. This feature is also helpful to determine if certain channels are under attack, since we can pinpoint with good precision where the fault occurred.

SIEM systems exchange mainly events and security information collected by the sensors. This data might be considered confidential in some organizations, while in others it can be of no concern. Since encrypting/decrypting every packet can introduce a reasonable performance penalty, and thus create delays that might affect timeliness, we decided to offer data confidentially on a optional basis. The application using the REB can indicate if it desires packets to be transmitted encrypted. In the this case, the other key shared between the source and destination is employed for the encryption.

3.2.2 Reliable and timely data delivery

TCP messages are delivered to their destinations if both the source and destination processes are correct and if the underlying communication network provides a *fair-loss* delivery. The Internet is one of such networks, in the sense that it makes an effort to transmit a packet through a route and then delivers it to the destination. It may happen that a packet is lost in-transit due to congestion or a crash of a network router. TCP solves this issue by segmenting the input stream and by transmitting one segment at a time, while waiting for the arrival of an acknowledgment of their reception. A retransmission occurs if it appears that the segment was lost, e.g., when the retransmit timer expires at the sender. As each segment is received and acknowledged, the receiver delivers them to the application. TCP does not offer any guarantee about when a particular message will arrive, though, it tries to optimize the communication to enforce the property of eventual delivery (without any upper time bound).

In REB, the messages input by the application are saved in a queue, and then they are split into several segments for dissemination. Unlike TCP, retransmissions based on

timers are avoided when possible so that the timeliness of a message is not affected. This is accomplished by preprocessing each segment with an encoder that applies an erasure code (also called a Forward Error Correction (FEC) code) to produce a number of packets. Depending on the code that is applied, if it is systematic or not, the resulting packets may contain the original data plus some repair information, or they may just have encoded data (see Section 3.5.3). The overall sum of the packets lengths is typically larger than the original segment, but it becomes feasible to reconstruct the segment even if some of the packets are lost.

The sender node disseminates the packets as they are produced by the encoder. It also starts a timer that should expire in case retransmissions have to be performed. However, retransmissions are typically avoided by adjusting the amount of repair capability of the code to the observed loss rate of the network. It may happen nevertheless that: (1) more packets end up being discarded, and therefore, the receiver is incapable of recovering the segment, or (2) the acknowledgement returned by the receiver is lost. When a retransmit timer expires, the sender encodes a few more packets based on the original segment (or selects some of the original packets), and forwards them to the destination. The expectation is that some of these packets will arrive at the receiver, allowing it to decode the segment. Once again, we need to initiate a new retransmit timer, now with a larger value.

The receiver accumulates the arriving packets in a receive queue. When enough packets of a given segment are available, the receiver attempts to decode them. In case of success, it returns an acknowledgement back to the sender. Otherwise, it waits for the arrival of an extra packet for this segment before trying again to decode. This process is repeated until the recovery of the original segment is accomplished. Depending on the network conditions, packets/segments may arrive and be decoded out of order. To address this issue, REB utilizes a selective acknowledgement scheme, to convey to the source information about which packets/segments have already arrived.

Communication in REB tries to be as timely as the network allows. By timely we mean that REB makes an effort to deliver data within the application defined deadline. This is important because events generated by the SIEM sensors maybe only useful for correlation if they arrive within their deadlines at the engine. The timeliness property of data delivery is provided by making use of multiple routes and some synchrony assumptions about the underlying network. Through the usage of a probing mechanism, it is possible to infer a quality metric about the overlay routes, which include estimated latency values. These latencies help selecting, based on the deadline of a transmitted message, the channels that are used to transmit the packets.

It can happen during particularly bad network conditions that it is not possible to deliver an application message within the specified deadline. Here, one could abort the transmission of this message, since it would probably be no longer of use to the receiver. This approach is however not the applicable to a SIEM. Consider the scenario where

a sensor produces an event that should be processed within a certain time period. If it arrives late, the event might not be properly correlated, but in any case its delivery can be useful from a perspective of forensic analysis. Therefore, even if a deadline is violated, the REB will still attempts to deliver the message.

3.2.3 Ordered and duplication-free data delivery

The Internet does not ensure an ordered delivery of packets. This occurs because different packets, sent by one source, may experience distinct delays when transmitted through diverse routes. Additionally, there is the possibility of spurious transmission of duplicate packets, which often happens due to rerouting algorithms in intermediate nodes. Despite these difficulties, TCP provides FIFO ordering at the delivery and also removes duplicates. This is accomplished by assigning to each segment a sequence number, which is used on the receiver side to order the segments. Furthermore, the sequence numbers are used to detect and discard duplicate segments.

In REB, the transmission of a segment corresponds to the dissemination of a fixed number of packets, which encode part of the original segment data and some redundant bytes. At the destination, the receiver decodes the original segment from a subset of the received packets.

Each segment has an associated sequence number (with 24-bits) that is incremented monotonically. These sequence numbers are employed to keep track of lost segments and to detect data duplication (accidental or from replay attacks). Packets also have a distinct sequence number (16-bits), which increases monotonically per segment (starting with 1). Therefore, a packet is univocally identified by carrying in the header a pair composed of the sequence number of the segment it belongs to plus its sequence number. Packets that arrive with the same identifiers are detected and removed as duplicates.

REB enforces FIFO order with the segment sequence number. A receiver node can deliver data in the right order by buffering complete unordered segments until all their predecessors have been given to the application. The amount of unordered data that is maintained in a REB node is managed through a receiver sliding window flow control mechanism. This mechanism prevents the receiver from accumulating too much unordered data, something that could be used by an malicious REB node to overflow the memory of the receiver. The sender is informed of how much empty space is still available inside the window, and packets that arrive beyond this space are discarded. The flow control mechanism is addressed in detail on Section 3.5.6.

We realize that there is a potential for an adverse effect caused by the simultaneous use of FIFO ordering and the assignment of deadlines to messages. The issue lies in the fact that a message with a deadline shorter than the one from its predecessor message, could eventually fail to be delivered on time due to the FIFO discipline. For example, it could happen that all segments comprising the second message are fully received before

the segments of the first message. Since the receiver node buffers the unordered complete segments until all their predecessors arrive, the second message would stay waiting in the queue and eventually have its deadline expire. One possible solution for this problem could be to use the deadlines to order the messages (which would become message priority values), but it is not entirely clear how issues such as message starvation would be resolved in a setting were REB nodes can be malicious. Our current approach is to keep the strict enforcement of the FIFO discipline, and as we get more experience with using REB, we may need to revisit this issue later on.

3.3 Sending and receiving data

The procedure of sending/receiving messages in REB encompasses a number of steps in order to enforce the various properties discussed in the previous sections, namely robust and timely guaranties with a simple to use application level interface. Since REB nodes are organized as an overlay network, the sender typically has several routes available for data transmission, which in most cases are expected to correspond to disjoint physical links. It is anticipated that most of these routes will fail independently with a reasonable probability. By forwarding data over a subset of the routes concurrently, the REB is able to tolerate some kinds of failures in a transparent way, and at the same time support the distribution of the network load over the alternative paths. Routes are selected using a metric that takes into consideration the observed Round Trip Time (or RTT) and loss rate in the recent past. The REB also uses erasure coding to create multiple blocks of data from the messages, which have in total a slightly larger size than the original message, and that are transmitted individually through the chosen routes.

The transmission process, which implements how individual messages are delivered from a source to a destination, is illustrated in more detail in Figure 3.2. When an application needs to send a message, it calls the REB interface and provides a buffer with the data, indicates the destination node and gives a deadline for the delivery (see also Table 3.4). Multiple calls can occur concurrently if the application has more than one thread. When the call returns, the application can re-use the buffer because it either has been transmitted or its contents have been locally copied to a send queue (named *segment queue*).

The REB maintains multiple segment queues, one per each active destination. Messages are copied to the queues in FIFO order. The procedure for selecting data from a queue for transmission involves the following steps:

1. If there is data to be transmitted, search the the various segment queues for the one with the message with the shortest deadline. Otherwise, keep on performing the other tasks (such as probing the links);

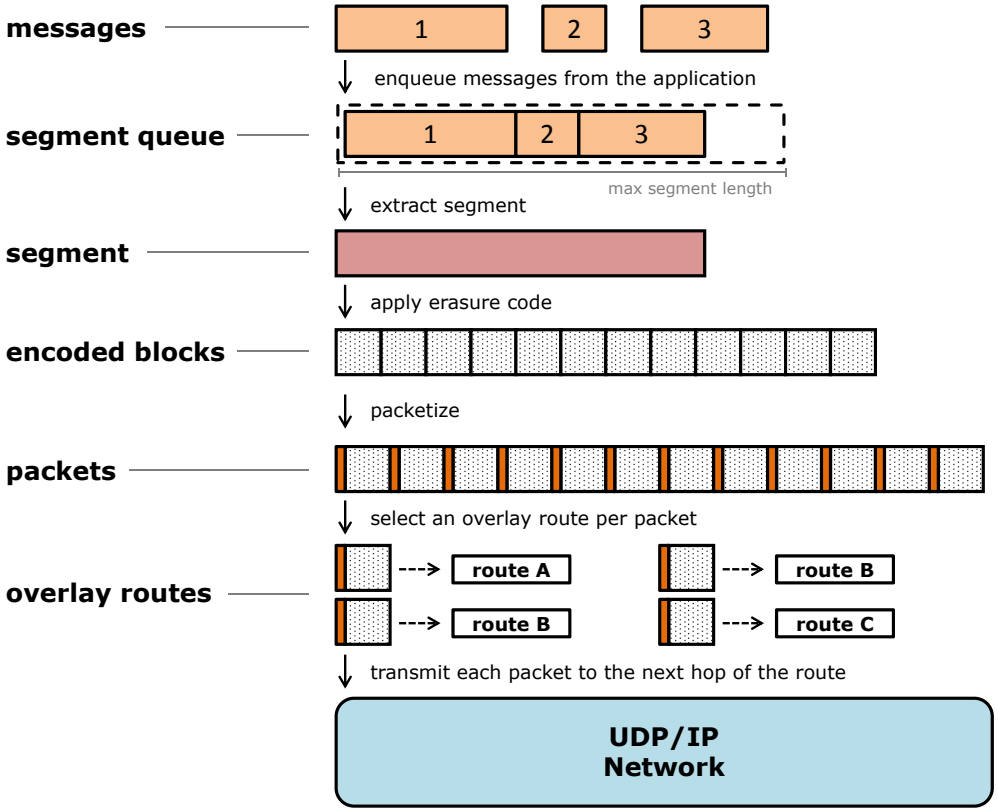

Figure 3.2: The data transmission process at a REB node.

2. Extract from the queue a segment of data:

   (a) If the queued data has a size larger than the *maximum segment length*, then create a segment with this length;

   (b) If the queued data has a size larger than the MTU (minus the headers) but less than the maximum segment length, then create a segment with the available data;

   (c) If the queued data is small (less than one MTU minus the headers), then (i.) use all data from this queue if there is no previously transmitted segment for the same destination that is still waiting for an acknowledgement; (ii.) otherwise, skip this queue and move to the next one.

The queue is mainly used for two purposes. First, in case the network is occupied with the transmission of older messages, a local copy is created so that the send operation can return, allowing the application to continue to run. However, if a message is too large and does not fit entirely inside the queue, it is split so that part of it fills the queue and the rest is scheduled to be inserted in when there is space again. Second, if the application normally sends small messages, the queue is utilized to accumulate more bytes. Ideally, one would like to increase efficiency by transmitting packets with a size approximately equal to the

MTU of the underlying network. Additionally, the coding algorithms require a reasonable amount of data for optimal execution. This amount of data is named the *maximum segment length* and its value has to take into consideration the size of the headers, the type of code and the network MTU between the sender and receiver. To prevent the creation of many small segments, but at the same time avoid delaying a message indefinitely, the procedure only allows a small segment to be transmitted concurrently to a certain destination (rule 2.c).

The extracted segment is then encoded by applying an erasure algorithm. The algorithm divides the segment in multiple blocks, and then processes them to produce the encoded blocks. If the algorithm is systematic (e.g., with Raptor codes [22]), then the first encoded blocks correspond exactly to the original segment and the remaining blocks contain repair data. When needed, the repair data is used at the receiver to reconstruct the missing blocks. On the other hand, in a non-systematic code (e.g., the LT code [12]) all encoded blocks have a mix of several blocks of the original data. If the segment is very small, it might be difficult to apply the most sophisticated algorithms, since often they are optimized for larger data sizes. In this case, it is more efficient to place the whole segment in a block, which is then replicated enough times to tolerate a certain number of failures.

Encoded blocks are then packetized by adding a header that contains, among other fields, the identification of the packet (sequence numbers of the segment and packet) and the final receiver. In order to maximize the reconstruction capability of the code, each encoded block should be placed in a distinct packet. This ensures that a packet drop in the network only affects one block, allowing the remaining ones to recover the lost data. In some cases, for efficiency reasons, it makes sense to include more than one encoded block per packet. This is only valid if two (or more) blocks fit inside the MTU of the network. This optimization should be utilized however with some care because it weakens the failure independence assumption on which erasure codes are based.

The sender next looks at the available routes to the destination, and selects a few of them that have a good figure of merit, allowing the segment to arrive within the deadline. A MAC is added to the header to let the final receiver detect integrity violations. In case an intermediary node needs to forward the packet, then a second MAC is also appended. The packet is then sent over the corresponding overlay channel, which translates in a transmission though the underlying network using UDP over IP.

On the receiver side, the node collects the packets arriving from the various channels as represented in Figure 3.3. The node is prepared to receive only a subset of the packets, since some of them may be lost in the network, while others can be corrupted. This second problem is detected with the included MAC, and the corresponding packet is deleted. The MAC is also employed for the protection of attacks where an external adversary (not controlling a REB node) may generate malicious packets, for instance, to confuse the segment reconstruction procedure or to make a DoS. Since the adversary does not share a

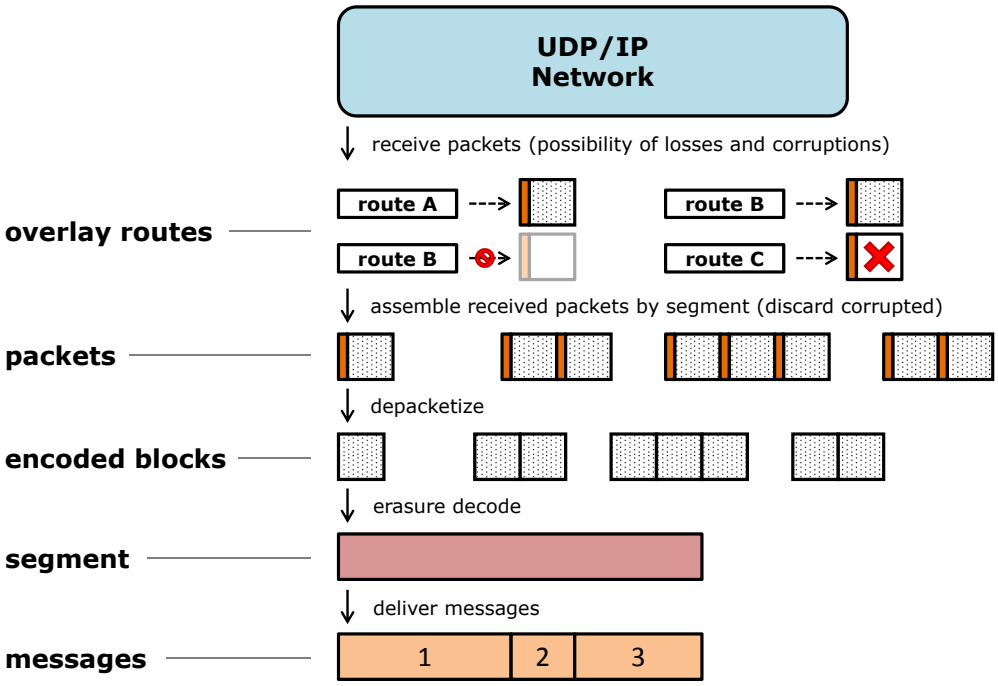

Figure 3.3: Scheme with the data receiving process.

key with the REB node, she is incapable of producing packets with the right MAC.

DoS attacks performed by a malicious REB node are addressed with a flow control mechanism (see Section 3.5.6 for details). When requested or piggybacked in the acknowledgements, the receiver indicates to the sender the amount of bytes that can be transmitted and that fall within the associated receive queue. Packets arriving when the queue is full are simply discarded. Additionally, packets with identifiers outside the expected range are also dropped, therefore, averting attacks that try to exhaust the memory by extending the local queues.

Packets can be received for the local node or to be forwarded to some other destination. In the first case, a few correctness checks are carried out, including duplication removal, and then the header is removed to obtain the encoded block. Next, the block is stored in the receive queue associated with the sender. Alternatively, when the node acts as a two-hop router, the packet is also checked and then enqueued to be transmitted to the final destination. To thwart starvation attacks caused by a malicious sender node, where it could try to delay the transmissions of this node, the packet is put at the end of the queue to wait for its turn.

Each encoded block is assembled accordingly to the particular segment that it belongs to. Since we are using erasure codes, we try to decode as soon as enough blocks have arrived. However, sometimes, it may happen that it is impossible to reconstruct the segment with the available blocks. When this happens, the receiver needs to wait for the arrival of more blocks, and then perform the decoding operation again. As segments are decoded,

they are stored in the receive queue waiting for the application to read them.

Initialization/Finalization calls	
init(local_id)	Initializes the REB object for the local node, on which the remaining calls are invoked. The parameter local_id indicates the textual identifier of the local node.
destroy()	Closes all communications and releases resources used by the REB object.
Communication calls	
send(buffer, size, destination, deadline)	Sends a message with size bytes from the data buffer to the specified destination node. The deadline value is a time value that is used to indicate the urgency of the message delivery.
receive(buffer, size, source)	Receives a message with size bytes and stores it in the provided buffer. The source value may indicate receipt from a specific node or from any source node (when a wildcard is provided).
setEncryptedMode(mode)	Sets the encryption mode for the communication. The mode value may be either *on* or *off*. The mode is changed when all pending data has finished being sent.
Information calls	
getLocalNodeID()	Returns the ID of the local node.
getLocalNodeAddresses()	Returns a list with all the IP socket addresses from the local node.
getRemoteNodesIDs()	Returns a list with the IDs of the remote nodes.
getRemoteNodeAddresses(id)	Returns a list with all the IP socket addresses from the remote node with the specified ID.
getRemoteNodeID(address)	Returns the ID of the remote node with the specified IP socket address.

Figure 3.4: Interface offered by the REB to the applications.

3.4 REB interface

REB is implemented as a Java library that can be linked with an application. It offers a relatively simple interface that contains the fundamental operations for transmitting data and some auxiliary methods for the application to collect information about the system. Table 3.4 gives an overview of the main operations of REB.

3.5 Communication mechanisms

This section describes some setup aspects and various mechanisms utilized by the REB. It offers an explanation about the erasure codes, as well as how multihoming contributes to the overall robustness of the communication. A mechanism for route probing is presented, supporting the inference of a quality metric for individual routes. This quality metric is then used by a route selection algorithm that is also discussed. To finish, we explain how the REB manages flow control.

3.5.1 Overlay network configuration and setup

The current version of the REB uses a static configuration for the overlay that defines the set of nodes that may participate in the communications (some of them may be down or disconnected). When a REB node starts up it is assigned an unique identifier, which is referred to as the *local ID*. The identifier is provided by the application that calls the startup interface of the REB (see Table 3.4).

Based on this local ID, a node can get the information about the whole overlay network by reading a few configuration files. The files are put in a predefined place in the local machine by the administrator of the SIEM. The following information can be obtained:

Network addresses A REB node receives packets in a specific IP address and UDP port. The ports can be different across the overlay, depending on the machine where the node is located. If a machine has multihoming, then several IP addresses are assigned, one for each physical connection. When this happens, the configuration file has the list of IP addresses that can be used;

Pair-wise shared keys Every pair of nodes shares a secret cryptographic key for secure communication. Each key is stored in a separate file that is statically distributed to the relevant nodes and not shared by any third party.

At start-up, a REB node is connected to no one. A connection is only established when the application requests a message to be transmitted to a specific destination node. Establishing a connection consists in resetting any previous state and in setting up two session keys. The former is necessary because nodes are allowed to crash and then later to be reinstated in the overlay. It could occur that the destination node had been exchanging packets with the source, and then the source had to reboot. In this case, during the connection establishment, the destination node would become aware of the problem, and therefore, it would clean information maintained on behalf of the source[2].

The handshake protocol that is executed between the two nodes is relatively simple. It consists of three messages protected with the shared key (with a MAC and encrypted). In the first message, the initiator indicates that it wants to create a connection by including a nonce N1 and the local IDs of both parties ($< INIT, ID_{initiator}, ID_{destination}, N1 >$). The destination node responds with a second message, which has a new nonce N2 ($< INIT\_RESP, ID_{initiator}, ID_{destination}, N1, N2 >$), and that allows the initiator to authenticate the destination. The last messages is transmitted by the initiator, to support its authentication at the destination ($< RESP, ID_{initiator}, ID_{destination}, N2, N1, conf\_info >$).

[2]The cleaning includes discarding pending packets and out of order segments, and other management data. Completely reconstructed segments that are stored in order in the receive buffer cannot be deleted because the sender might have the expectation that they will eventually be delivered to the application. This is important in the case that the connection needs to be reestablished when the segment sequence number reaches its maximum value.

The destination node only resets the connection when the $RESP$ is correctly received, and it uses information in $conf\_info$ to determine exactly how this operation should be performed. Nonces are created by concatenating two values, a locally generated random number with a high resolution timestamp (up to the microsecond).

It can happen that one of the handshake messages is lost in the network. To address this problem, the sender of the message is responsible for its retransmission until the other side responds. So, for instance, if message $INIT$ is dropped, the initiator should periodically resend it until the following handshake message arrives. If the destination node sees a duplicate $INIT$, this could either indicate that $INIT\_RESP$ was lost or delayed. In this case, it simply waits for its retransmission timer to expire, which would cause $INIT\_RESP$ to be resent, or for the arrival of $RESP$ that would conclude the handshake.

The loss of the last message, $RESP$, is recovered in a slightly different manner. From the point of view of the destination node, it cannot distinguish the case when $INIT\_RESP$ or $RESP$ are dropped by the network. Therefore, it keeps retransmitting $INIT\_RESP$ until a $RESP$ arrives. At the initiator, when a duplicate $INIT\_RESP$ is received, it resends the $RESP$ message. In all situations, handshake messages are only retransmitted a certain pre-defined number of times. When the limit is reached, and the connection attempt is aborted and an error is stored in a log file.

When the overlay is being setup, it may occur that two nodes attempt to start a connection simultaneously. In this case, both would send concurrently the $INIT$ message, and both would receive the peer's $INIT$ as the response. To solve this issue, we use a simple arbitration procedure, where the node with the largest ID aborts its connection attempt, and follows the handshake launched by the other side by responding with a $INIT\_RESP$.

An adversary could take advantage of the handshake protocol to attack the REB communications. For example, she could replay an old $INIT$ message to fake a reboot of the initiator to force a connection reset. Since in general the destination node cannot distinguish a replay from a valid connection attempt, it starts the handshake protocol by responding with the $INIT\_RESP$. However, it continues to process the packets arriving from the initiator as usual, ignoring for now the $INIT$. Since the adversary does not know the shared key, she can not produce a valid $RESP$. Therefore, after a number of $INIT\_RESP$ retries, the destination node forgets about the connection.

In another example attack, the initiator could send an $INIT$ message, and then the adversary could replay an old $INIT$ of the node. As a consequence, the destination would receive two valid but different $INIT$ messages from the same node. In this case, we again use a simple arbitration procedure, where the handshake corresponding to the $INIT$ carrying the nonce with the largest timestamp is the one that is executed, and the

other is disregarded[3].

As a final attack example, when a node initiates a connection to a destination, where the $ID_{initiator} > ID_{destination}$, the adversary could replay an old $INIT_{old}$ message from the destination to the initiator. When this happens, the initiator applies the arbitration procedure for a simultaneous connection, and stops its handshake. Then, one of two things can happen. First, the destination node receives the original $INIT_{original}$ and responds accordingly with the $INIT\_RESP$. As the initiator gets the $INIT\_RESP$, it checks that the message carries the $N1$ from its $INIT_{original}$ and a $N2$ with a timestamp larger than the one in $N1$ from $INIT_{old}$. This provides evidence that an attack occurred, and therefore, the initiator returns to the original handshake and completes it with a $RESP$. Second, the adversary could also remove the $INIT\_RESP$ and all its retransmissions, meaning that it has complete control of the routes between the two nodes. In this case, the initiator will retransmit the $INIT\_RESP$ corresponding to $INIT_{old}$ a number of times and then abort the connection, which an appropriate action given the attack power of the adversary.

A node simply ignores a message for which the corresponding previous handshake message was not received (if a $INIT\_RESP$ or a $RESP$ arrives without having sent the $INIT$ or $INIT\_RESP$, respectively). Additionally, since a node only resends a message a predefined number of times, this prevents denial of service attacks where an adversary keeps replaying the most recent handshake messages (either $INIT$ or $INIT\_RESP$) to force the destination to perform retransmissions.

At the end of the handshake, the two session keys are produced to secure the communication between the nodes. One key provides authentication and data integrity by being used to generate the MACs included in every transmitted packet, and the other key provides confidentiality by encrypting data. The formula for producing the keys is the following:

$$K_{MAC} = hash(SharedKey, N1, N2, ID_{smaller}, ID_{larger}, "M")$$

$$K_{Encryption} = hash(SharedKey, N1, N2, ID_{smaller}, ID_{larger}, "E")$$

In the formulas, $SharedKey$ is the preconfigured shared key between the nodes, $N1$ and $N2$ are the nonces from the handshake, and ID are the identifiers of the nodes. The identifiers are placed in a deterministic order, first the smaller ID, so that both sides produce the same keys. Strings "M" and "E" are used to differentiate the two keys.

The new keys substitute the old ones when the handshake finishes. Packets from the previous connection may still be in transit when this occurs. These packets will be

[3]Here, we are working under the assumption that the adversary can not change the clock of the initiator to a later time, and then collect an $INIT$ message with a larger timestamp. If this assumption is violated, then a new shared key needs to be setup between the two nodes by the SIEM administrator. Notice, however, that even in this case the adversary cannot complete the handshake protocol and deceive the two parties.

discarded upon arrival because the MACs are no longer valid. This mechanism has the benefit that prevents packets from an older session from confusing the receiving algorithms.

3.5.2 Multiple paths and multihoming

A REB node can typically reach a destination through many different overlay routes. If the REB is able to determine which routes are behaving erroneously, and picks alternative paths for data transmissions, it is possible to tolerate failures in the network. Of course, these measures are only effective if the failures do not completely cut all communications.

Since REB uses one-hop source routing, the available paths are the following: first, there is the direct link from the sender to the receiver; second, since any other node can act as an intermediate router, each one of them defines an extra path. Overall, in a REB deployment with n nodes, there are at least $n - 1$ paths connecting every pair of nodes.

Facilities may be interconnected via multihoming, i.e., by two or more distinct physical links (e.g., a REB node could have a pair of network interfaces and would be connected through two different ISPs). Since these links usually only share a minimum amount of resources, their failure can be considered independent in many scenarios (e.g., a DoS is performed in one of the ISPs). REB takes advantage of multihoming to increase the number of available overlay paths, allowing a node to overcome the failure of one of its links by exploring alternative routes.

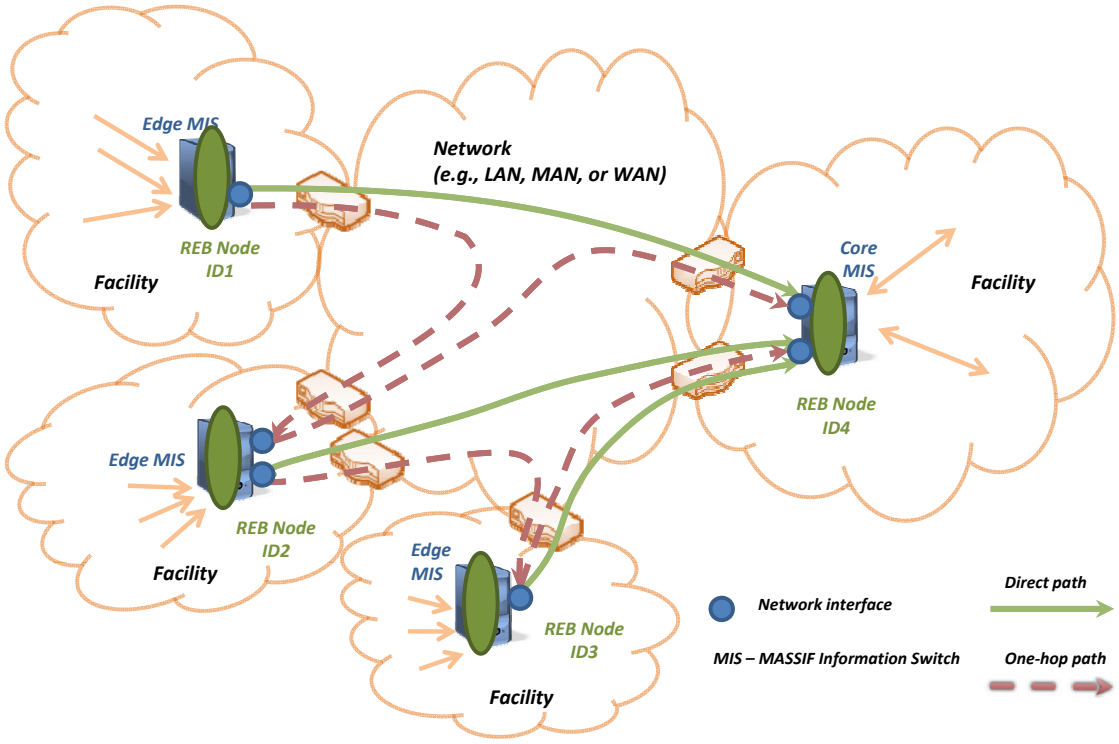

Figure 3.5: Multihoming in REB.

The number of network interfaces directly influences the quantity of overlay paths a local node may have at its disposal. Figure 3.5 shows an example overlay network configuration that makes use of multihoming in order to provide a diversity of links. REB nodes with identifiers ID1 and ID3 have a single network interface, while nodes ID2 and ID4 have two interfaces. Therefore, node ID2 can reach node ID1 through two direct links, and can send packets to node ID4 over four direct paths.

The exact number of overlay routes that exist can be calculated in the following way. Let y be the number of interfaces on a certain local node and w the number of interfaces on another remote node; in total, there are $y \times w$ available direct paths between the nodes. Two-hop paths include an extra intermediary node and make use of the same interfaces as in the direct paths, as well as the interfaces of the intermediate node (the packet can arrive in one interface and then leave from any of the available network interfaces). Let z_i be the number of interfaces on a certain intermediary node. Then, the number of paths that can go through this intermediate are $y \times w \times z_i^2$. In total, the number of available paths between the two nodes are:

$$Total\ paths = \sum_{i=1}^{n-2} y \times w \times z_i^2 + y \times w$$

Returning to the example from the figure, the total number of paths between nodes ID1 and ID4 is equal to 12. There are two direct paths; eight two-hop paths through node ID2; and two two-hop paths through node ID3. For overlays with many nodes, the growth in number of paths could become challenge if all of them were used in the communications. However, only a subset of the paths is actually employed by a node to transmit data, and these are picked based on their quality metric.

3.5.3 Multipath transmission and erasure codes

A REB node sends packets concurrently over a few of the available channels to the destination. However, data transmission through multiple channels per se is not sufficient to achieve robustness in the communications. In fact, even a single channel behaving erroneously (e.g., losing packets) is enough to prevent the original data from being reconstructed. Therefore, using multiple concurrent channels can actually degrade the reliability of the whole communication.

Two possible approaches to recover from losses are (1) the retransmission of the packets at a later time, or (2) the concurrent transmission of several copies of the packets over different channels. The first solution has the advantage of minimizing the amount of data that is send, at the cost of delaying the delivery of the packets (since retransmissions occur after a timeout). The second approach has the opposite characteristics.

In REB, we use erasure codes to both decrease the amount of transmitted data and to minimize the delays in case of losses. Before disseminating a segment, it is split into k

equal sized blocks. These blocks are then encoded to generate N encoded blocks. The encoding process ensures, with very high probability, that the reception of any K blocks is enough for the recovery of the full data, where K is slightly larger than k (and less than N). Therefore, if there is a limited quantity of losses in the network (less than $N - K$ blocks are dropped), then it is possible to reconstruct the original segment in a timely manner, without requiring retransmissions. However, it may happen that the network is behaving worse than expected, and only a less than K blocks arrive. In this case, further communication will be required, either by retransmitting some of the encoded blocks, or by producing and sending a few extra encoded blocks[4].

REB currently uses Fountain Codes [14, 12], or rateless erasure codes, that can infinitely encode the data. In our current implementation, we resort to LT Codes [12], but in the future we intend to replace these by the newer and more efficient RaptorQ Codes [13]. In LT codes, an encoded block is created by XORing a few of the original blocks. Two random functions are called, one to determine the number of blocks that should be XORed, and the other to select the actual blocks. Decoding is performed gradually, as the encoded blocks arrive at the receiver. In each step, it is checked if the new encoded block allows the recovery of an original block, and if this is the case, this knowledge is further propagated to decode other blocks. Eventually, when enough encoded blocks reach the receiver, it is possible to reconstruct the segment.

At the source, if sufficient data is available in the send queue, then the segment size is selected in such a way that every encoded block completely fills a packet. This means that an encoded block should have EB_{lenght} bytes, which is equal to MTU minus the size of the headers (added by the REB, UDP and IP). Since the original blocks have the same dimension as the encoded ones, then the segment size should be $k \times EB_{lenght}$. This ensures an efficient utilization of the network, and also that a packet drop only affects a single encoded block, something that is typically assumed by the codes.

Erasure codes are, however, able to recover from bursts of encoded block losses. Therefore, if the segment is small, one can put a few encoded blocks in the same packet to reach a dimension similar to the MTU (Note that the random functions used by the LT codes are devised for particular values of k, and therefore it is not possible to simply decrease k without affecting the properties of the code). This way one can keep the network efficiency, at the cost of potentially losing more encoded blocks. For tiny segments that fit in a single MTU, there is no advantage of employing coding algorithms. In this case, we simply replicate the segment over a few packets and send them concurrently.

At the destination, the arrival of the encoded blocks triggers the decoding process, and when the whole data is decoded, a confirmation is sent back to the source node. If decoding is unsuccessful, then the source is responsible for retransmitting some of the

[4]The exact solution depends on the erasure code being used. The second approach that generates new encoded blocks has the virtue that the sender does not have to know which blocks arrived correctly.

missing encoded blocks. This makes the delivery reliable since it is assured to happen in the presence of correct processes and a best-effort network (like the Internet).

3.5.4 Segment and packet identification

Segments have to be delivered in FIFO order to the application, so it is necessary to univocally identify them to allow a proper organization at the receiver, in case they arrive or are decoded out of order. When transmitting a segment, it is also required to identify each packet to determine which encoded blocks have been received. As a result, each packet carries a unique ID on its header that identifies the segment the enclosed blocks belong to, as well as their position inside the segment.

A packet ID comprises a concatenation of two sequence numbers, resulting in a total length of 40 bits. Out of those 40 bits, the first 24 represent the segment sequence number and the remaining 16 correspond to the packet sequence number inside the segment. Both sequence numbers start at 1 and are incremented monotonically, having an upper bound of $2^{24} - 1$ and $2^{16} - 1$, respectively.

The first upper bound, for the segment sequence number, is expected to overflow eventually if the REB is used over long periods of time. When that happens it is necessary to reset the connection between the source and destination nodes, so that these sequence numbers may be safely reused without the danger of introducing corruption of data caused by replay attacks. Basically, after the connection is reestablished, new shared keys are derived, and therefore packets carrying the previous sequence numbers will not be accepted as the MAC is invalid. The second upper bound, for the packet sequence number, is not expected to overflow because REB limits the number of encoded blocks generated from a segment.

The space size for the segment sequence number was defined using the information provided by the MASSIF use case scenarios [15]. For example, the Olympic Games scenario, at its pick load, produces around 12 million events per day before aggregation. If these events were to be transmitted in separate segments, something highly unlikely because there is typically aggregation of events at the collectors, then the connection would only have to be reestablished with a frequency less than once a day.

3.5.5 Acknowledgments and Retransmissions

During a transmission, the destination node receives the packets and stores the data in memory until enough encoded blocks are available for decoding. After the successful segment reconstruction, the node sends an acknowledgment back to the source and makes the segment readable to the application.

The usage of erasure coding and multiple overlay routes offers a good level of robustness against lost packets. Despite this, the achieved reliability has a probabilistic nature,

meaning there is always a (small) chance that the segment cannot be rebuilt at the destination. One reason for such failure is an insufficient number of encoded blocks reaching the destination node, caused by too many packets being dropped in transit. Another reason is that even when K encoded blocks arrive, the decoding algorithm might be incapable of recalculating the original data (with a small probability). To overcome these problems, the source needs to retransmit packets if it does not receive an acknowledgment within a predetermined time.

The network may reorder the packets being disseminated. Additionally, the use of multiple routes in the overlay may also contribute to the arrival of out of order packets, as the paths can have diverse transmission times. Therefore, as unordered packet arrival will potentially occur often, it is advisable that the destination node keeps these packets in buffers instead of dropping them. In REB, a separate buffer is managed for each source at the receiver, and is named *receive queue*.

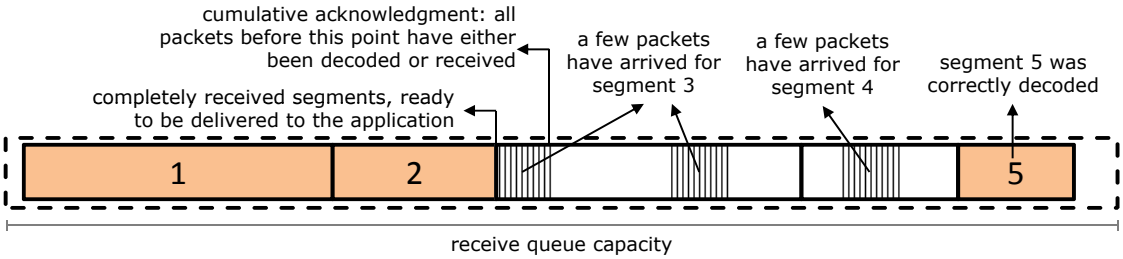

Figure 3.6: Example of a receive queue.

The receive queue has a fixed length, and allows for the storage of several segments. It can contain segments completely decoded and packets belonging to various segments that have been partially received (see an example in Figure 3.6). As the application reads the stored data (in FIFO order), the corresponding space is freed so that further packets can be accommodated. Packets are placed in the queue in the right place accordingly to their identifier (segment sequence number + packet sequence number), and consequently there might holes in the queue so that out of order packets can be added at a later time. Packets that would need to be stored beyond the queue limit are simply dropped (as the queue is not extended to save them; however, as the flow control mechanism is used exactly to prevent these packets from being transmitted, this can only occur if the sender is malicious).

REB informs the source about which packets/segments have been correctly received with a *selective acknowledgment* (SACK) (a mechanism somewhat inspired in TCP [17]). A SACK contains for each partially received segment a list of pairs of packet IDs, which define continuous intervals of stored packets (e.g., if only packets five to nine have arrived of a certain segment S, then the range is $[S.5, S.9]$). To avoid always having to explicitly define the intervals for previously completed segments, the first pair of IDs represents a *cumulative acknowledgment* that confirms the reception of all previous segments and

packets (including the indicated packets itself). This optimization is possible because segment sequence numbers are incremented monotonically and cannot be reused during a session. In the queue of the example figure, the cumulative acknowledgement appears at the end of the first packets of segment 3.

REB implements a few simple rules with regard to the management of the receive queue and the return of acknowledgements:

1. The receive queue places the packets/segments in their expected position with respect to their identifiers.

2. The receive queue does not store packet/segments beyond its maximum size. Therefore, out of bound packets are dropped.

3. The receive queue never garbage collects packets of the segment that is currently being received, and that have been acknowledged to the source.

4. The receive queue can garbage collect packets and segments that are beyond the segment currently being received, even if they had been acknowledged to the source.

5. Selective acknowledgments may only confirm the reception of part of packets/segments in the queue as there is limited space in a SACK packet. The packets/segments that should be acknowledged are the ones that appear first in the receive queue.

In the example of Figure 3.6, the segment that is currently being received is 3. Therefore, its packets will never be garbage collected. The saved packets for segment 5 or even segment 5 may be removed if the node needs to reclaim their space.

The source node starts a timer with a predetermined duration after the transmission of the last packet of the segment (in the flow control section, we will look into the case where there are more segments to be send). Ideally, the duration is defined in such a way that allows for the receiver to decode the segment, and return the acknowledgement. Figure 3.7 shows an example of a transmission in this normal scenario.

If the timer expires before an acknowledgment is received, then the network or the receiver had a problem. For example, some of the data packets were dropped and the segment could not be reconstructed, or there were delays in the network or/and receiver that made the acknowledgment arrive later. Since the source does not know the cause of the problem, it sends a special packet with no data (referred to as a *ping packet*) to the destination node to ask about which packets have been received so far. When the destination node receives a ping, it immediately sends a SACK back to the source. The ping packets are transmitted periodically, until either an acknowledgment arrives or the sender gives up (and returns an error). With the reception of the SACK, the source node starts to

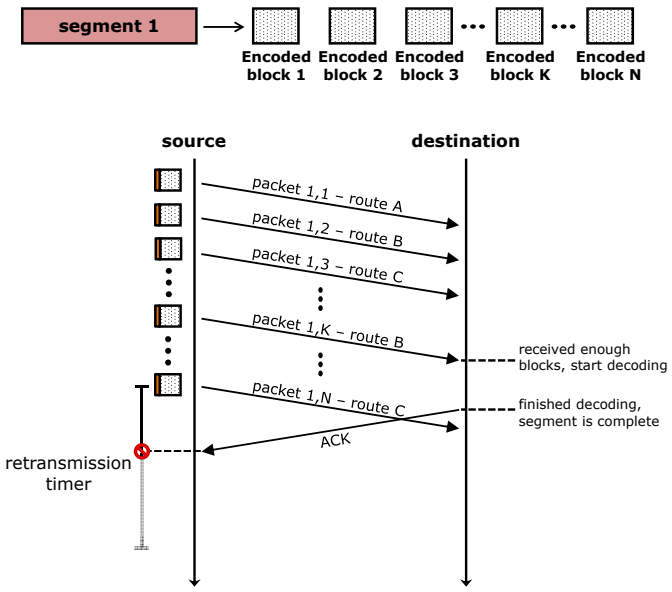

Figure 3.7: The common scenario for transmissions.

retransmit all missing packets. Figure 3.8 show an example scenario of a retransmission caused by lost packets.

Following the example of TCP, the value of the retransmission timer of REB is calculated based on the estimated RTTs and variations [21]. However, in REB one needs to account for multiple overlay routes, which have independent quality of service metrics. Since a node keeps information about each overlay route RTT as well as an average of the RTT variation, it is possible to use this information to select a reasonable value for the timer.

In REB, it was decided to take a conservative approach for the calculation of the timeout. We use the worse values for the expected RTT and RTT variation of all the paths were packets were transmitted. Additionally, since the decoding process at the receiver may take a non-negligible interval, the calculation of the retransmission timer also takes into account an estimation of this period. The decoding time depends on the type of erasure code being used, and specifically on its decoding algorithm complexity. To simplify the estimation, we assume that it takes approximately the same interval to decode as to encode, and therefore we measure its value at the sender. As we get more experience in using the REB, one may need to devise a more accurate way to make this estimation.

Let $RTTest$ and $RTTvar$ be the values for the estimated RTT and RTT variation, respectively, of the route with highest expected RTT ($RTTest + RTTvar$), and let δ be the estimated processing time for decoding. The retransmission timer value is calculated as:

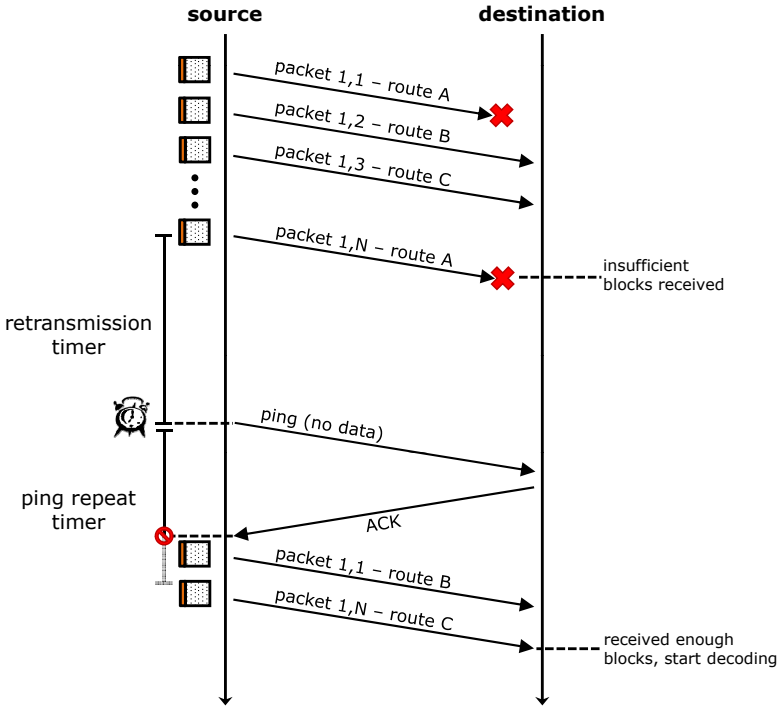

Figure 3.8: A retransmission caused by a high packet loss.

$$T_{ret} = RTTest + max(1, 4 \times RTTvar) + \delta$$

.

Three important issues must be considered when addressing retransmissions. The first is that they can potentially cause a delivery of a segment to expire its deadline. In general it is not possible to solve this problem because the network is lossy and might be under attack. However, to minimize the probability of this event, the REB combines the usage of multiple routes and erasure codes to make the need for retransmissions very unlikely.

The second issue is related to the reliability of SACK delivery. SACK can be lost due to an accidental corruption in the network or because of an attack that causes its deletion. In order to maximize the probability of SACK arrival, it is sent through all the routes that were used by the source node when transmitting the data packets. A destination node must therefore keep a record of these routes.

The third issue concerns acknowledgment loss, duplication and/or reordering, either of accidental or malicious nature. For example, an attacker may try to replay previous acknowledgments to a source node, trying to force unnecessary retransmissions. Even in a non-malicious scenario, the re-ordering of two acknowledgements could make the source think that some of the previously received packets were garbage collected, therefore, they would eventually need to be retransmitted. To solve this sort of problems, the source applies the following procedure when an SACK arrives:

- Compare the acknowledgment information in the new SACK with the last accepted SACK with regard to (a) the value of the cumulative acknowledgement, and (b) the ranges of correctly received packets in the segment that is currently being received.

  - If any of the two indicates that less packets have arrived, then discard the the new SACK;
  - Otherwise, update the current knowledge of what has been received accordingly to the new SACK.

3.5.6 Flow control

When a source node sends data to a destination, it may have several segments ready for transmission. In this case, it would be interesting if the node could disseminate packets from multiple segments before receiving an acknowledgment for the initial one. This has the benefit of allowing source nodes to do useful work while the destinations are busy decoding packets, or while the SACK is being forwarded through the network. However, the increased transmission capability needs to be bounded, otherwise too many packets may reach the receivers, which can cause memory exhaustion on their machines (or too many packet drops). To address the problem, REB implements a flow control mechanism to limit the transmission rates between nodes.

The rate of a transmission is dictated by the capacity of the receive queue at the destination, but more importantly by the rate of data consumption by the application. The receive queue holds (in order) the latest completely received segments until they are consumed by the application. As a result, the queue must have at least a capacity large enough to hold one encoded segment with maximum length. To tolerate out-of-order reception, blocks from subsequent segments are also buffered inside the queue. If those segments are received before the previous ones, they are decoded but stay in the queue until the earlier segments are read (in order) by the application.

The portion of the receive queue that starts immediately after the last in order complete segment (or the whole queue if it is empty) can be used to store further packets. When managing this space, we give priority to the next transmitted segment because it will be read by the application right after the existing segments. Therefore, the packets of this segment, which we call as the *segment currently being received* (see Section 3.5.5), are never reclaimed by the garbage collector in case of lack of memory in the node. The packets from the following segments can, however, be deleted.

The *receive window* is defined as the free space in the receive queue. We include in this space all packets/segments that are more recent (i.e., with higher segment sequence number in the session) than the segment that is currently being received. Therefore, the storage of these packets/segments is considered provisional.

A source node keeps an informed view of the receive window, based on information that is returned by the destination. The current view sometimes does not reflect the actual window state because packets may be lost, but in normal network conditions it tends for the right value. This view is required by the source node because the data transmission rate is constrained by its value (i.e., the the source can only send packets that fit within the available space). To keep the value accurate, the destination node includes the size of the receive window in each acknowledgment.

When a source node has outstanding encoded segments for a given destination, it starts transmitting them block by block in a FIFO order, (ideally) each one inside a distinct packet. The transmission procedure is implemented in such a way that the initial outstanding segment is given a higher priority than the rest. The procedure steps are as follows:

1. Let $winSize$ the currently perceived value for the receive window size;

2. For the first outstanding segment, obtain the constants: N is the total number of encoded blocks; K is the typical number of blocks required by the destination node to decode the segment with high probability; and l is the length in bytes of each block;

3. Set variable D to the total number of delivered blocks from the first segment so far (0 at the beginning of the transmission);

4. Obtain the number of undelivered blocks from the first segment that fit inside the window, as $Z = winSize/l$ (integer division);

5. If the receive window size is insufficient to hold all the undelivered blocks from the first segment, that is, if $Z < N - D$, then:

   (a) Send Z undelivered blocks from the first segment and keep a record of those transmissions;

   (b) Begin a periodic transmission of ping packets (packets with a data of size 0, as described in Section 3.5.5), to force the destination node to respond with an acknowledgment;

   (c) Try to send segments to other destinations.

6. Otherwise, if there is space in the receive window:

   (a) Send the remaining undelivered blocks from the first segment, that is, send $N - D$ blocks;

   (b) Start the retransmission timer (as described in Section 3.5.5);

(c) Update the receive window size, as $winSize = winSize - l \times (K - D)$ (note that it is K instead of N);

(d) If there are outstanding segments beyond the first one, apply a similar procedure and send as many blocks as the receive window allows; otherwise, try to send segments to other destinations.

Notice that when sending extra blocks from the first segment (beyond the initial K), we take an optimistic approach to update the receive window (rule 6.c). This is done because in most cases K blocks are enough to decode a segment. As a result, the destination node needs to manage the receive queue accordingly, being prepared for the fact that K blocks might not be enough and that extra blocks might need to be stored ($N - K$ blocks). This means that in some cases, a destination node may discard received blocks from the next segments (and possibly even decoded segments), if not enough space is available in the queue. This is reflected on the way acknowledgments are understood by the source, where a newer SACK may "rollback" the delivery status of subsequent blocks/segments. Figure 3.9 shows an example of this scenario. Note that thanks to the efficiency of the erasure codes, it is expected that such scenarios occur rarely in practice.

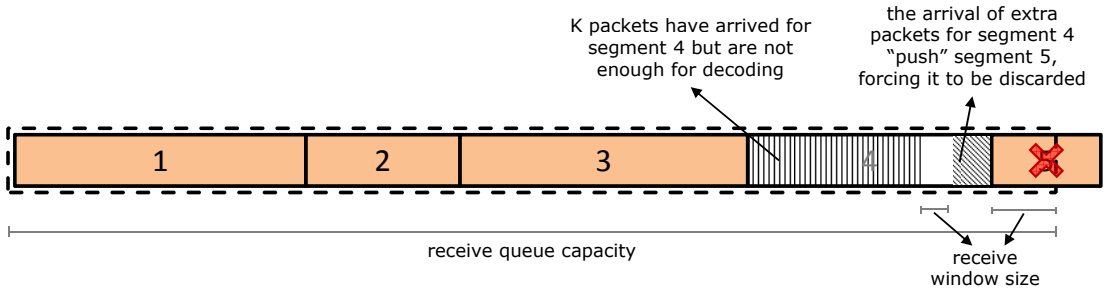

Figure 3.9: Example scenario where a segment has to be discarded from the receive queue to get space for storing packets of the segment that is currently being received.

On the subject of window size updates, it is possible for a duplicate acknowledgment (spurious or forced by an attacker) to give an incorrect view of the window, forcing unnecessary retransmissions. This can happen when a destination node sends acknowledgments which roll back the information about the delivery status of subsequent segments (see above). This situation arises because even though it is possible for a source node to detect an ordering on the acknowledgments by analyzing the delivery status of the first segment (which never rolls back), different acknowledgments that only differ on the subsequent segments cannot be consistently ordered. This roll back behavior however is expected to occur only on rare occasions, so this particular scenario does not significantly affect the communication. Besides, note that it only has a small impact on the communication progress because the first segment continues to be delivered correctly.

On the subject of ping packet transmissions, there seems to be a possibility for a DoS attack on destination nodes causing them to transmit a large number of acknowledgments in a row. To circumvent this problem, destination nodes only respond to ping packets if their period is greater than a fixed amount of time.

It is also worth mentioning that when receiving an acknowledgment, the advertised window size could be too small (smaller than the size of a packet). To avoid this problem, the receiver only advertises windows with a capacity that allows the transmission of packets with a considerable size. Currently, this minimum capacity is MTU.

3.5.7 Route probing and selection

The timeliness of the communication is based on the assumption that the appropriate selection of overlay routes will, with high probability, result in the delivery of segments before specified deadlines. Therefore, it is required to periodically estimate the quality of service of the routes, in order to have a continuous informed and updated knowledge about the best available paths.

Concerning metrics of quality of service, the fundamental one of an overlay route is the expected Round Trip Time (RTT). For a source node, it is both important to know how much time will take for a given segment to reach its destination, and also how long it will take before the respective acknowledgment arrives back. One should not forget that a source is only capable of continuing the transmission of buffered segments after the acknowledgment is received, with the respective update to the window size. Another relevant metric is the loss rate of a route, which influences the effective time necessary for a packet to arrive to a destination (if a packet is lost, then the transmission time can become the interval for the timer to expire plus the retransmission through the network). Therefore, we do not use the loss rate directly in the choice of the best routes, but as a way to penalize negatively the RTT value of a route. This way, routes with high loss rates are viewed as being slower and so fall behind others with better latencies and/or lower loss rates.

In order to estimate the RTT values of its routes, a source node utilizes a probing mechanism that is activated periodically, causing the destination node to return back information about the delivery delays and packet losses. Given the current size of SIEM deployments, the REB overlay may have a few hundred nodes. To keep the overhead of probing small, we adjust the probing frequency to the usefulness of the routes – routes that are used regularly are probed more often. Furthermore, if a node normally communicates with a certain destination, then the routes towards that node are checked more frequently.

REB uses two different approaches to manage probing traffic efficiently: a) a source node transmits probe requests through unused routes to trigger immediate replies with probing data by the destination nodes (a *pull approach* by the senders); b) a destination node also initiates the transmission of probing data through some routes whenever a

segment is fully received or whenever enough probing information is available to be transmitted (a *push approach* by the receivers). Consequently, a route that is used recurrently will have more probing traffic being conveyed by the receiver. Routes that are never utilized are only checked when the source decides to send a probe through them. This allows for a fast recovery of the routes being currently employed for the main communication (should they suddenly become attacked), while keeping the knowledge about idle routes updated over time.

The probing mechanism is intrinsically connected to the transmission of acknowledgments, that is, probing information is delivered to a source node inside acknowledgment packets that are returned from a destination node. As we have seen before, acknowledgments are transmitted when a segment is completely received, as well as when a source node explicitly requests them through the use of packets with zero-length data (referred to as ping packets). We can see an immediate parallel with the push and pull approaches defined earlier. Probing requests are nothing more than ping packets and probing information is immediately transmitted per receipt of these packets. There is one difference concerning acknowledgments in this case, though, which is that acknowledgments may be transmitted earlier, that is, before a segment is completed. This may happen if enough probing information is available to fill its space in a packet, forcing a transmission of that information along with an acknowledgment. Note, however, that the receipt of an earlier acknowledgment by a source node does not affect its main communication since acknowledgments are only processed if they acknowledge new data (see Section 3.5.5).

However, sending probing data only in specific moments such as when segments are completely received, as opposed to transmitting this data when packets arrive, could at first seem to affect the precision of the RTT estimation at the source. Adding such transmission of acknowledgments every time any packet is received through some route, though, would increase the network traffic, which could interfere with the main communication by delaying it and causing processes to stall. It was thus necessary to find a mechanism that kept the extra traffic to a minimum, but at the same time offered a fine granularity on the probing of individual packets (information about their loss rates and individual latencies).

To achieve the desired granularity of having probing information about individual packets during a normal transmission, a destination node stores the arrival times of every received packet and includes those times inside the acknowledgment. At the same time, a source node stores the departure times of each packet it transmits, as well as the identification of the route that was used. When an acknowledgment arrives, the source node calculates the RTT value of each packet using the saved departure time and the announced arrival time. It then uses that sample RTT value to estimate the expected RTT value of the route that was used to transmit the packet. Since REB nodes might not have their clocks synchronized with very high precision, a source node cannot simply calculate the latency

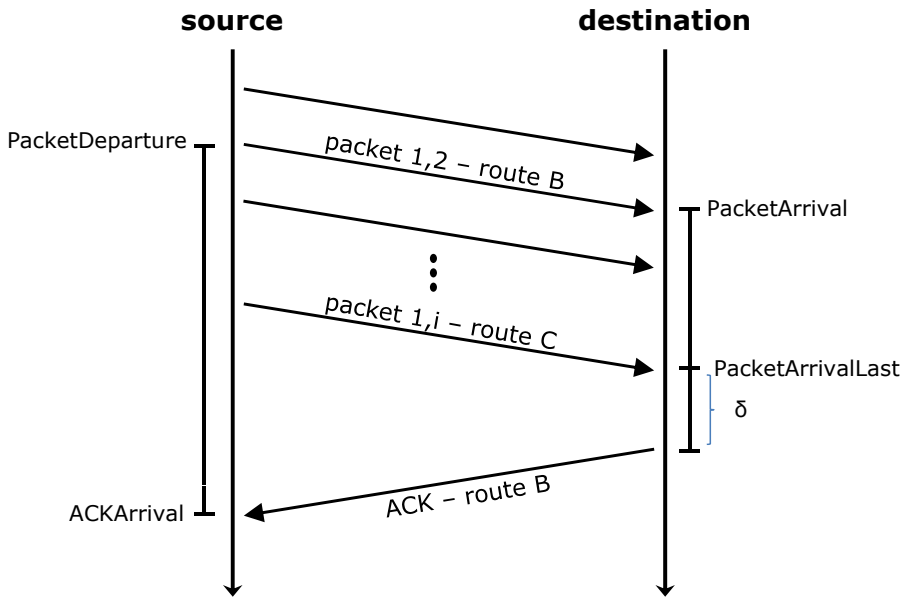

Figure 3.10: Example scenario where the RTT is estimated for packet with ID (1,2).

of a packet by subtracting its departure time from its arrival time. Instead, a source node obtains the time of the acknowledgment arrival, and for each reported packet inside:

1. Measures the elapsed time between the moment the packet was transmitted from the source *PacketDeparture* and the moment the acknowledgment arrived back *ACKArrival*;

2. Obtains the elapsed time between the moment the packet arrived at the destination node *PacketArrival* and the moment the acknowledgment was transmitted from there;

3. Subtracts the first elapsed time from the second and hence obtains the sample RTT value.

Note that in the second step, the source does not receive the instant when the acknowledgement was sent, but gets the moment when the last packet arrived *PacketArrivalLast* that allowed the segment to be decoded. Since it may take some time for the decoding operation to conclude, the instant of the acknowledgement transmission is estimated by adding δ to *PacketArrivalLast* (recall that δ is approximately equal to the decoding time). Presenting it in a formula, the sample RTT value is taken as (see Figure 3.10 for a graphical representation):

$$\begin{aligned} RTTsamp =& (ACKArrival - PacketDeparture) \\ & - (PacketArrivalLast + \delta - PacketArrival) \end{aligned}$$

It is important to refer that each acknowledgment must be transmitted through all the routes from where the reported packets were received. In turn, for every acknowledgment copy it receives, the source node must only measure the sample RTT values from the packets that were transmitted through the same route as the acknowledgment. This is required because otherwise it would be possible for the latency of a route to affect the estimation of the RTT of another.

The way REB calculates and keeps an expected RTT value per route is built upon the well tested algorithms employed by TCP [21]. As mentioned before, the expected RTT value corresponds to the sum of an RTT estimation and an RTT variation. Both metrics are averaged over time following an exponentially-weighted moving average, which keeps an history of past values but gives more weight to recent values over old ones. For each route, both metrics are computed as follows:

- If no sample RTT values have been taken, then:
  - $RTTvar = 0$
  - $RTTest =$ a predefined large value
- When the first sample RTT value has been taken, then:
  - $RTTvar = RTTsamp/2$
  - $RTTest = RTTsamp$
- When an additional sample RTT value has been taken, then:
  - $RTTvar = (1 - \beta) \times RTTvar + \beta \times |RTTest - RTTsamp|$
  - $RTTest = (1 - \alpha) \times RTTest + \alpha \times RTTsamp$
  - Note that here, $RTTvar$ is updated using the value of an $RTTest$ from the previous update.

Following the advice in [21], α is set to $^1/_8$ and β to $^1/_4$. However, these values presuppose a modus operandi for transmitting data that comes from TCP, and may not be ideal in REB because of the different transmission semantics. This subject will require a further understanding in order to ascertain whether different values for α and β make more sense in the context of REB.

Loss rate is another metric which is used to affect the perceived quality of the overlay routes. In order to detect lost packets, source nodes inspect the IDs from the packets reported in the probing information. If there are "holes" within the listed IDs, a source node assumes the respective packets were lost in transit.

The loss of an acknowledgment can also affect the perceived loss rate of a route. In order to identify the loss of such acknowledgments, each one carries in the probing

information the ID of the last packet reported in previous acknowledgments. A source node assumes then that every unacknowledged packet which has an ID inferior to the one indicated above is lost. However, since acknowledgments are transmitted through all multiple routes, the chance of all being lost is reduced. Every time a packet is deemed lost, the $RTTest$ value of the route the packet was sent through is affected negatively in a fixed amount λ.

The algorithm for selecting the routes that should be used for transmitting the packets uses the paths with best quality of service (i.e., $RTTest$). At this point, this algorithm has been kept relatively simple. For a given destination, the source picks R routes for transmission. Of these routes, some of them should be direct links to the destination and the remaining should have an intermediary node. This ensures a reasonable level of diversity among the chosen routes, which can be beneficial in case of attacks. The quota for direct links (up to $R/2$) is first filled in with the best direct routes. For the remaining slots are picked the two-hop paths that have also the best quality of service.

Chapter 4

Implementation and Evaluation

This chapter describes several aspects related to the implementation and installation of the REB. It also includes an evaluation of the REB in a demonstration prototype network.

4.1 REB library installation

The REB is available as a Java library that provides an API for other Java applications. There are no external dependencies (no third party libraries) required when using the REB, only the set of REB classes which come pre-built inside a Jar archive and some configuration files. The source code is also provided and an Ant file exists for automated class building. In addition, there is a tool for generating random secret keys for pairs of REB nodes.

4.1.1 Build procedure

To compile or use the REB library, it is required version 1.7 of the Oracle JDK. To use the automated building mechanism, it is also required version 1.8.4 of the Apache Ant tool.

The installation procedure of REB is relatively simple because all classes come pre-packaged in a Jar file. To build the REB classes using Ant, open a terminal to the directory where the REB was extracted (a file named "build.xml" is available in the directory) and run the command `ant`. To clean the build run the command `ant clean`.

To use the key generation tool, you need to first build the class files as explained before, and then set the following environment variables:

- `REB_HOME="path to where REB was extracted"`
- `PATH=${REB_HOME}:${PATH}`

Then run the command `genkeys <REB node 1> <REB node 2> <...>`, which generates one secret key for every pair of the indicated nodes and stores the keys on a different file per node.

4.1.2 Configuration files

The current version of the REB uses a static configuration for the overlay that defines the set of nodes that may participate in the communications (some of them may be down or disconnected). When a REB node starts up it is assigned an unique identifier, which is referred to as the *local ID*.

Based on this local ID, a node can get the information about the whole overlay network by reading a few configuration files. The files are put in a predefined place in the local machine by the administrator of the SIEM. The following information can be obtained:

Network addresses A REB node receives packets in specific addresses (IP address + UDP port). The ports can be different across the overlay, depending on the machine where the node is located. If a machine has multihoming, then several IP addresses are assigned, one for each physical connection. When this happens, the configuration file has the list of IP addresses that can be used;

Pair-wise shared keys Every pair of nodes shares a secret cryptographic key for secure communication. Each key is stored in a separate file that is statically distributed to the relevant nodes and not shared by any third party.

The first file is *nodes.cfg*, and it includes an identification for every node consisting of a string and a set of IP socket addresses. Copies of the nodes file have to be placed in every machine that hosts a REB node. Listing 4.1 shows the syntax for a nodes configuration file. The second kind of file is *keys_<id>.cfg*, containing the secret symmetric keys that a specific local node shares with the remaining remote nodes. There must be an individual key file per node, each containing as many keys as the number of the remaining nodes - the tool `genkeys` can be used to assist in the creation of these files. Listing 4.2 shows the syntax for a key configuration file.

4.2 Architecture of a REB node

REB is provided as a library that offers a communication service to Java applications. The interface for the application allows for communication with multiple remote nodes over the same object. This contrasts with a unicast socket-oriented communication interface, where each socket is used in a distinct point-to-point communication, independently of other sockets. However, such an interface would not be useful in REB because the local node transmits messages, not only to their destination, but to intermediary nodes as well, which forward them to their destination. Furthermore, in the future, communication in REB may support multicast where the local node can send a message to multiple destinations, and in that sense the current interface already provides the necessary multiplexing logic for such a communication, easing the extension process.

```
-- REB nodes configuration file --

Each line configures a REB node by defining an identifier and a set of
network addresses. The syntax is:
[identifier] [address_1] [address_2] ... [address_n]

An identifier is composed of the prefix "eng" or "sen" followed by a
positive integer. Note that the identifier prefix serves only to
remind the user of the REB, the type of the machine (SIEM engine or
SIEM sensor) that hosts the node.

A network address is composed of an IP address/hostname + port,
separated by a colon (:). It is possible to define multiple addresses
per node since REB supports multihoming.

Examples:
eng1 1.2.3.4:20001 1.2.3.5:20002 1.2.3.6:20003
sen1 4.3.2.1:30001 4.3.2.2:30002
sen2 8.7.6.5:40001
```

Listing 4.1: Syntax for the file nodes.cfg

A REB node runs on the same process of the application, executing most of its functionality on different threads and keeping application messages on internal queues until they are delivered. For this reason, an application should only require at most two threads to interact with the local node (one for sending data and another for receiving). On communication operations, the interaction of an application thread with the local node is handled differently according to the kind of operation. In a sending operation, an application thread indicates the required ordering of message delivery to the node, while the node manages the traffic rate by restricting the amount of bytes that are transmitted at a time (which may block the application thread as feedback). In a receiving operation, an application thread dictates the rate of incoming message consumption, thus affecting the flow control applied by the local node, which is perceived by the remote sending node and consequently by the sending application thread. Altogether this provides an effective feedback mechanism which adapts to the application needs.

4.2.1 Main Components

The main components of a REB node are divided according to their natures:

Execution: *Threads* running on the process for the local node;

Spatial: Storage *queues* which enforce buffering of messages/packets and a communication medium for the threads;

Temporal: Communication *sessions* and *timers*.

```
-- Shared keys configuration file --

The name of the file must indicate the name of the local node. The
syntax for the file name is "keys_<id>.cfg", where <id> is the
identifier of the local node. For example, a file named
"keys_sen1.cfg" lists the secret keys of the local node sen1.

Each line configures a secret key shared by the local node and a
specific remote node. The syntax is:
[identifier] [key]

A remote node is specified by defining its identifier (must not be the
identifier of the local node).

A secret key is a string with a 256-bit number in hexadecimal format
(64 characters).

Examples:
sen2 0123456789abcdef0123456789abcdef0123456789abcdef
eng1 fedcba9876543210fedcba98fedcba98fedcba9876543210
```

Listing 4.2: Syntax for the file keys_<id>.cfg

Together, these components form a state machine capable of handling events from the application and the network, and providing the expected communication among the local node and other remote nodes.

Threads

A REB node uses three threads internally (plus one in special cases) to run its state machine (note that this precludes any Java Virtual Machine (JVM) threads that handle operations such as automatic garbage collection). Those threads are a *Dispatcher Thread*, a *Receiver Thread* and a *Sender Thread*.

The Dispatcher Thread is responsible for handling events from application threads, events from the Receiver Thread, and events from timer expirations (timeouts). It is also responsible for producing events to any other thread, including an application one.

The Receiver Thread is responsible for handling events from the network and for producing events to any other thread, including an application one.

Finally, the Sender Thread is responsible for handling events from the Main Dispatcher Thread and the Receiver Thread. It only produces events to the network.

There is one more thread which is used when the application requests a special type of sending operation, namely an asynchronous one. This thread is called the *Asynchronous Dispatcher Thread* and handles all the asynchronous send requests. From the point of view of the Dispatcher Thread, the Asynchronous Dispatcher Thread is just another application thread because it provides messages using the same execution logic.

An application thread is also considered part of the state machine since it is both an

event source and an event destination.

Queues

The main functions of the REB node's queues are to provide an ordered delivery of messages and a flow control mechanism applied to the communication. Messages are copied to the queues in FIFO order and the queues have statically defined capacity bounds for limiting flow. Three types of queues are distinguished: a *segment queue*, a *local receive queue* and a *remote receive queue*. There is one of each of these queues for each remote node that is configured at the local node (i.e., each local "remote node object" contains its own set of queues).

Segment queues are accessed by application threads and the Dispatcher Thread. Application threads copy messages to these queues and the Dispatcher Thread retrieves the buffered messages into segments to be transmitted.

Local receive queues are accessed by application threads and the Receiver Thread (additionally, the Dispatcher Thread may also access these queues for session maintenance). The Receiver Thread copies received segments to these queues and application threads retrieve bytes from these queues in order to deliver received messages.

Remote receive queues are accessed only by the Dispatcher Thread. They represent a view of the opposite (local) receive queues for the corresponding remote nodes, maintaining a delivery (or acknowledging) status of individual transmitted segments.

There is one more queue which is not associated to any remote node in particular. It is the queue utilized by the Sender Thread to receive events from the Dispatcher and Receiver Threads. Here, the events are simple packet send requests with an associated remote address and local address (a REB node is capable of multihoming). Packets that are provided by the Dispatcher Thread have the local node as the source, while packets that are provided by the Receiver Thread have a specific remote node as the source. This latter case occurs when the local node acts as an intermediary node in the communication between two other nodes by forwarding received packets to their destinations. The usage of this queue in this situation is essential in preventing DoS attacks that attempt to cause starvation of local outgoing packets.

Sessions

Sessions are required for successful communication between two nodes since they authenticate both parties in the exchange of messages. A session provides a full duplex communication (messages may be transmitted in both ways simultaneously) but does not allow a "half-open" or "half close" situation like in TCP. For this reason, when a session is established between two nodes, it must be maintained at both sides even though only one of the nodes is transmitting data. Similarly, when a session is closed, both sending and receiving is disallowed until a new session is established.

A session has an associated symmetric key which is exchanged securely between two nodes during session establishment. This key is used to provide the necessary authentication and additional message integrity validation using MACs which are appended to transmitted packets.

Sessions are initiated automatically by a REB node whenever the application tries to send a message to a remote node. Therefore, an application has no control over the sessions and they are handled internally by the Dispatcher Thread. Currently there is no explicit session close mechanism implemented, so the only way for sessions to be closed is for one of the session owners to crash (or simply destroying the local REB object). Eventually, at the the other side of the session, the surviving REB node will detect a session timeout and will close it properly. However, a session may also be closed by a reset at a local node if the remote node crashes and recovers immediately in time to request a new session before the previous one times out at the local node.

Timers

There are three types of timers within a REB node. All three have specific instances associated to different remote nodes. Two of them are associated with the transmission of data and the other is associated to a communication session.

In a data transmission scenario there is a retransmission timer which is activated after the successful transmission of the first outstanding segment. There is also a probe retransmission timer which is activated after each transmission of a probe packet (see Sections 3.5.5, 3.5.6 and 3.5.7).

Finally, there is an additional session timer which is used to detect when a session times out. Every time the local node receives an acknowledgment from a particular remote node, it updates its session timer with that node. To keep sessions alive in the absence of data being transmitted, the local node sends keep-alive packets (probe packets) periodically which trigger the transmission of acknowledgments by the remote node.

4.2.2 Two interaction examples

In this section we show two examples of interaction between the main components of a local REB node.

The first interaction example is for a message sending scenario. Figure 4.1 shows the example. Here an application thread is copying a message to the segment queue associated to the indicated remote node (given the destination node ID). The local object for the remote node also contains the respective remote receive queue which is accessed exclusively by the Dispatcher Thread. The figure shows the dispatcher thread extracting a segment from the segment queue and placing it on the remote receive queue for transmission. Inside the remote receive queue, a segment is transmitted block by block, after

the segment is encoded using LT Codes. Each block is transmitted through a different overlay route, which is indicated to the Sender Thread by specifying different IP socket addresses (remote and local).

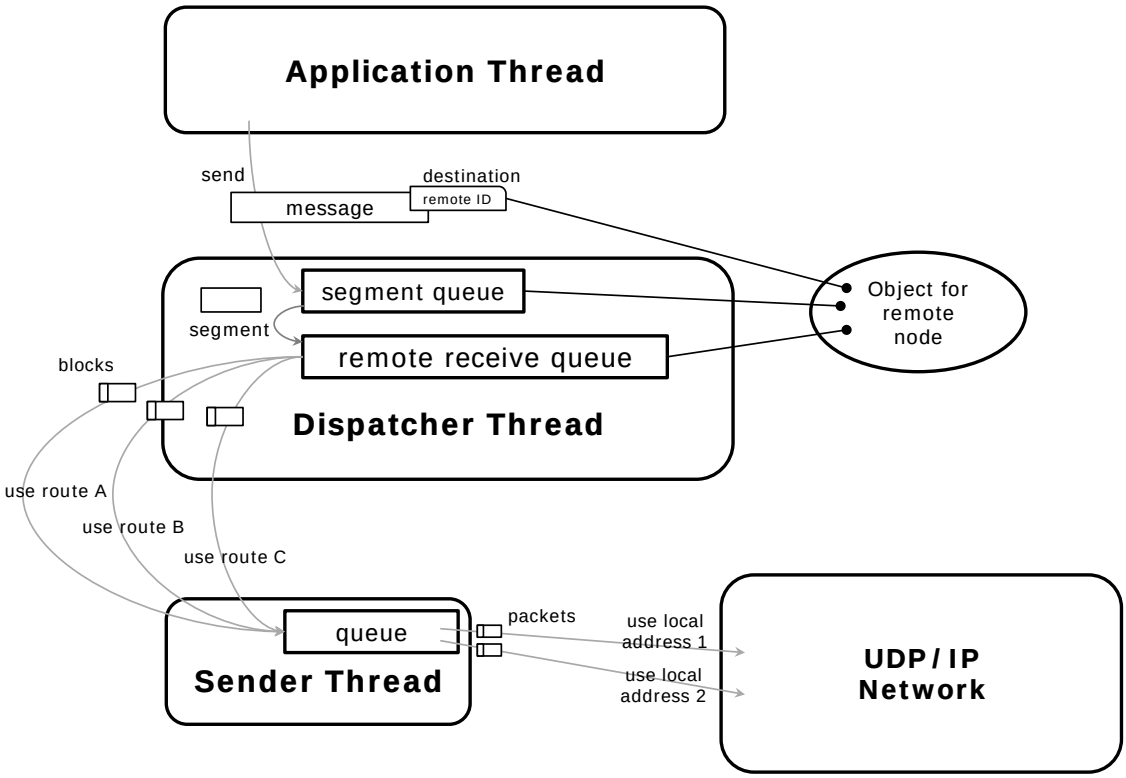

Figure 4.1: Interaction of the components in a sending scenario.

The second interaction example shows three different receiving scenarios simultaneously. Figure 4.2 shows the example. Packets that arrive from the network are aggregated by the Receiver Thread using a demultiplexer (in Java this is a `java.nio.Selector` object, which is the equivalent of the network call *poll* from C). Three received packets corresponding to three different receiving scenarios are illustrated in the figure. The first packet (the top right one inside the Receiver Thread) contains a destination which is not the local node. This is the forwarding scenario in which the local node has the role of an intermediary in the communication of two other nodes. Here, the packet is simply given to the Sender Thread, which will eventually transmit it to the correct destination. The second packet (the one received acknowledgment (ACK)), is addressed to the local node, so it is given to the Dispatcher Thread so it can process it in the right remote receive queue (associated to the object for the remote node identified in the packet source). The last received packet (the one on the left containing data) is addressed to the local node, so it is placed by the Receiver Thread in the appropriate local receive queue. This example also shows an application thread accessing the same receive queue to deliver received messages.

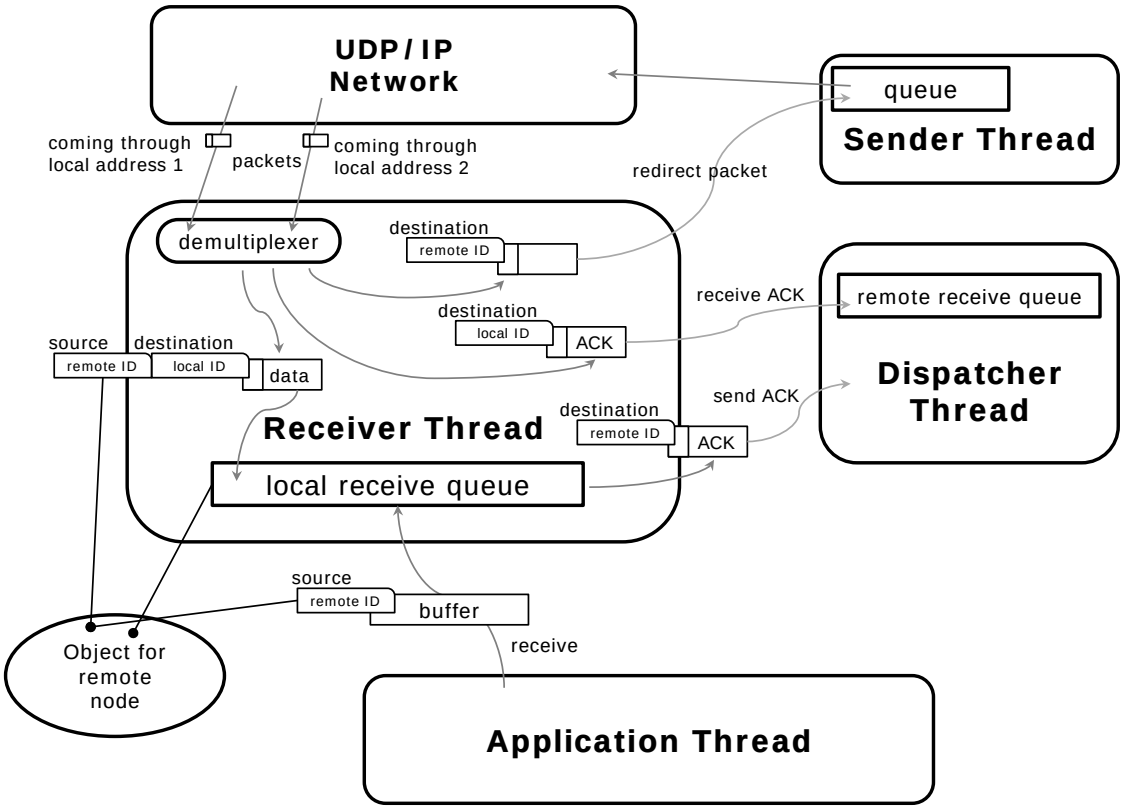

Figure 4.2: Interaction of the components in multiple receiving scenarios.

4.3 Design of a REB node

In this section we explain the design of the REB library, showcasing how some of components of a REB node are implemented.

4.3.1 Node identification

Each REB node has a unique identifier, which is represented by a string of characters formed by the prefix “sen” or “eng”, followed by a positive integer number. Examples of nodes’ identifiers are “sen4” or “eng2”. The prefix is an indicative of the type of SIEM machine that hosts a certain node (a sensor or an engine), but does not otherwise make the node execute any differently. Each node is also associated to a set of IP socket addresses (IP address and port), which are used to identify the nodes on the underlay network. The size of a node’s address set depends on its host, namely the number of available network interfaces, so a REB node will always have at most as many addresses as the number of existing network interfaces of its host machine. Every REB node knows the address set of every other node it knows about, including the order of the addresses in the set which is the same everywhere (the order is defined in the nodes configuration file addressed in Section 4.1.2).

A node’s identifier consists of a string of characters, as mentioned before. This textual

representation is only used by the user of the REB, though. Internally, REB nodes compress a node's identification in a 16-bit integer and a particular address from its address set in an 8-bit integer, in order to minimize the overhead on data transmissions.

Of the 16 bits that represent a node's identification, the least significant 15 represent a positive numerical value, while the 16th bit defines the type of the node (bit 0 defines an engine and bit 1 defines a sensor). This effectively limits the number of distinct REB sensors and distinct REB engines to 32767 each.

The 8-bit integer that represents a particular address is defined as the ordinal number of an address in the set, starting at 1 (an address set's order is universal for all nodes). Each node's address set is therefore hard limited to 255 distinct addresses.

Whenever it's necessary to define a *null* identifier (which does not identify any of the REB nodes), the 16-bit integer is zeroed. The same applies for the 8-bit integer representation of a "null" address.

4.3.2 Packet structure

REB uses UDP as the underlying transport protocol, so data is transmitted inside UDP datagrams. A UDP datagram indicates its size in the UDP header and is transmitted entirely in one IP datagram, unless IP fragmentation occurs (however we restrict the maximum UDP datagram size to minimize this situation). Because REB uses UDP, we say that the communication in REB is *message oriented*, meaning that a transmitted REB packet by one node is either received in its entirety by another node or not at all. For this reason, a REB node can immediately start processing a received packet from a UDP socket, without needing to reassemble the transmitted packet from multiple received fragments.

A REB packet contains a route header which must always exist since it is required by a node to know the exact overlay routes used in the transmission of data. A packet also has one or two MACs in order to authenticate the involved parties and to provide message integrity verification. Figure 4.3 illustrates the general structure of a REB packet. The route header is 11 bytes wide and each MAC has 32 bytes. Depending on the type of data that is transmitted, the remaining contents have a specific structure inside a packet. Types of data include encoded blocks, tiny segments, acknowledgments and handshake messages.

Route header

The route header of a REB packet contains the information about a specific overlay route, which includes a source node, a destination node, and optionally an intermediary node. It also contains bit flag values which are used to identify the type of data that follows the header.

Figure 4.4 shows the structure of a packet header. The node ID fields are 16-bit values which represent numerical forms of the node identification strings (such as "sen4" or

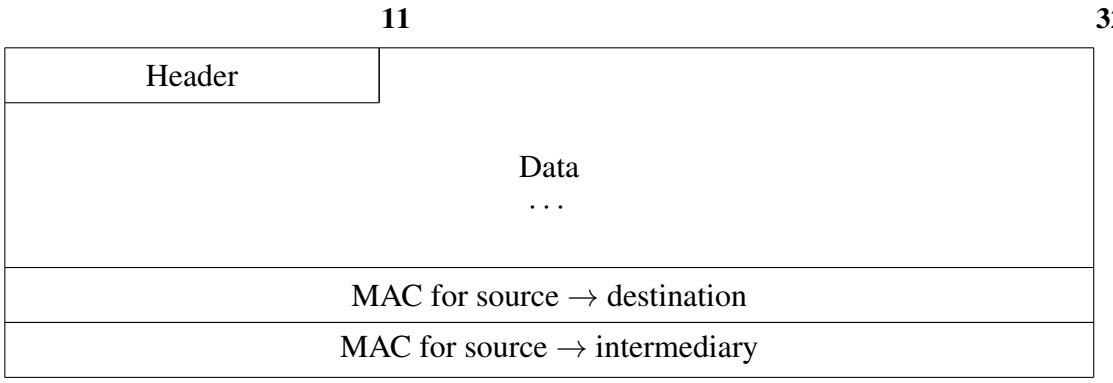

Figure 4.3: Structure of a REB packet (sizes in bytes).

"eng2"). The IDs are ranged from 1 to 65,535, and a value of 0 represents a *null* ID which is used in the intermediary ID field if a direct route is used. The address fields are 8-bit values which represent ordinal numbers associated to a node's IP socket address. The ordinal value of an address of a particular node corresponds to the position of that address in the line defining the node in the nodes configuration file (see Section 4.1.2). The address ordinals are ranged from 1 to 255, and, like the IDs, a value of 0 represents a *null* address which is used in the intermediary address fields if a direct route is used. The ranges for the IDs and address ordinals limit the number of nodes and the number of addresses per node that can be configured in a REB network (see Section 4.3.1 for more details about this).

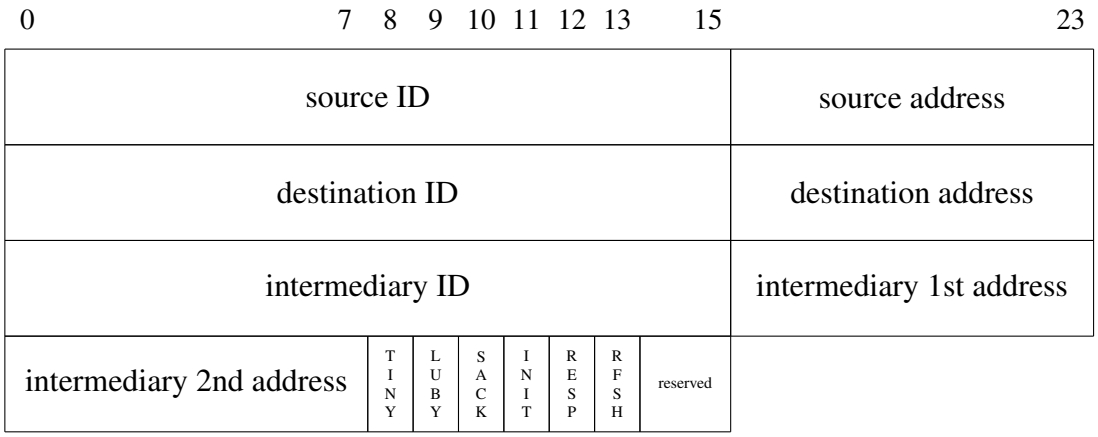

Figure 4.4: Structure of a packet header.

The flags field is 8-bit wide and contains several bit flags that identify the type of data that follows the header. The flags used in regular data transmission are:

- `TINY` - if only this flag is set, a node will process the data as a tiny segment, that is, a segment with a length sufficient to fit entirely inside a REB packet;
- `LUBY` - if only this flag is set, a node will process the data as a block from a segment

encoded with LT (Luby Transform) Codes;

- `SACK` - if only this flag is set, a node will process the data as a Selective Acknowledgment;

The flags `INIT` and `RESP` are used to identify handshake messages. Handshake messages are of three kinds, *INIT*, *INIT_RESP* and *RESP*. Flags `INIT` and `RESP`, when each being the only set flag, represent an *INIT* and a *RESP* message, respectively. When both are set (and no more flags), they represent an *INIT_RESP* message. The flag `RFSH`, shorthand for "refresh", is used to distinguish two kinds of a *RESP* message.

Segments and blocks

Depending on the size of an extracted segment from a segment queue, the segment will have a different type. If its size is less than or equal to the Maximum Transmission Unit (MTU) size minus the headers (of IP, UDP, and REB) and MACs, then the segment is transmitted entirely inside one UDP datagram. This type of segment is referred to as a *tiny segment*. If on the other hand the segment size is greater than this limit, it is first encoded using LT Codes, generating a fixed number of blocks (N), and then transmitted block by block, each inside one UDP datagram. This type of segment is referred to as an *encoded segment* and each REB packet is said to carry an *encoded block*. A probe packet that is used to force a destination node to transmit an acknowledgment is simply a tiny segment with zero length data.

Whether a REB packet is carrying a tiny segment or an encoded block, is distinguished by the flags `TINY` and `LUBY`. However when a destination node receives a packet, it needs to know the size of the segment being transmitted as well. It also needs to know which specific segment or block is being transmitted so that it may be able to put it in the right place in its local receive queue.

Figure 4.5 shows the structure of a segment header plus data inside a packet. The first field is a 32-bit number which indicates the size in bytes of the original segment, that is, the segment being transmitted in the packet if it is a tiny segment or the segment before encoding if the packet contains an encoded block. In both cases it is possible to know the size of the data inside the packet using the original segment size - if it is a tiny segment, the data size is trivially the original segment size, if it is an encoded block, the data size is calculated by the destination node in the same manner as the source node when it's generating encoded blocks. The original segment size is particularly helpful when an encoded block is being transmitted because it allows the destination node to remove any padding added by the LT Codes after decoding the original segment. The second field is a 24-bit sequence number identifying the transmitted segment and the third field is a 16-bit sequence number identifying the encoded block within the encoded segment (if the data is a tiny segment this sequence number is zeroed and not used).

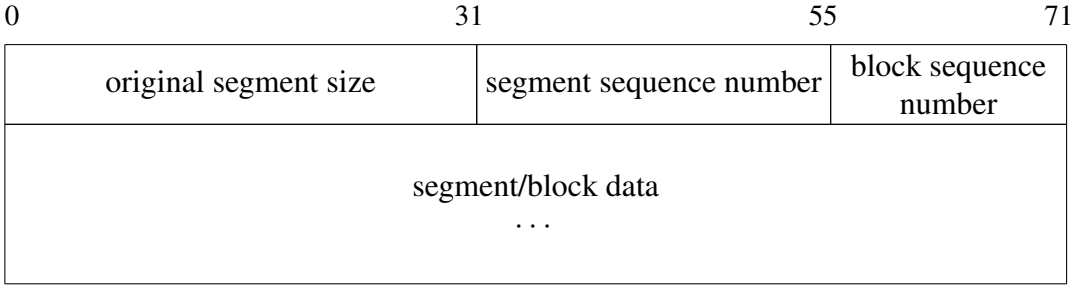

Figure 4.5: Structure of a segment header plus data inside a packet.

The LT Codes algorithm splits original segments in a fixed number of *source symbols*, and produces *encoded symbols* with the same size as the source ones. An encoded block carries an encoded symbol and a pseudo-random seed value used by the destination node to obtain identification of the source symbols that were encoded. Figure 4.6 shows the structure of an encoded block. The seed value is represented as a 64-bit value inside the packet.

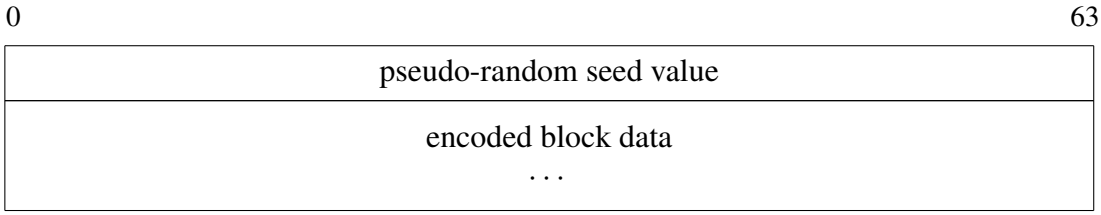

Figure 4.6: Structure of an encoded block inside a packet.

Selective acknowledgment

A selective acknowledgment is transmitted whenever a destination node receives a complete segment (either by finishing decoding one or by receiving a tiny segment), or whenever it receives a probe packet (a tiny segment with zero-length data). The acknowledgment includes information about the status of the received segments in the local receive queue, and about specific blocks within encoded segments. It also includes the window size of the queue, so that the source node may limit the transmission flow.

Figure 4.7 shows the structure of a selective acknowledgment inside a packet. The first field is a 32-bit number indicating the window size. The second field is a 24-bit number indicating a cumulative sequence number which acknowledges the receipt of all the segments with sequence numbers below or equal to that number. The third field is an 8-bit number indicating the number of segments that have status information in the rest of the acknowledgment. Each segment information contains a 24-bit sequence number and an 8-bit number indicating the number of block pairs included in the segment information. If the segment being acknowledged is fully received (either is a tiny segment or a decoded segment), this number is always zero. For each encoded segment, the status

information of its blocks is indicated as a list of block pairs. Each of this pairs contains two 16-bit sequence numbers identifying the first and last blocks of a contiguous range of acknowledged blocks. Since the size of an acknowledgment is restricted by the maximum permitted data size inside a packet, not all received segments/blocks may be able to be acknowledged even if they are fully received. For this reason, a selective acknowledgment includes status information of the segments in increasing order of sequence number, starting by the undelivered segment with lowest sequence number. Block pairs are also indicated in increasing order of sequence number.

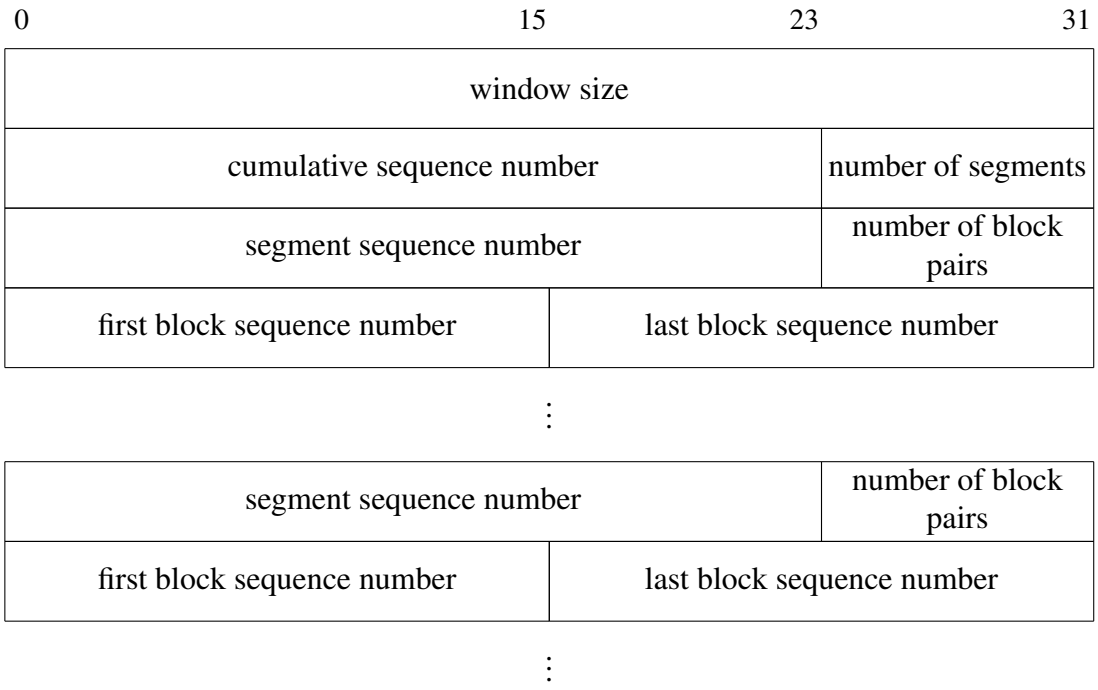

Figure 4.7: Structure of a selective acknowledgment inside a packet.

Additionally, an acknowledgment that is transmitted after a different one may contain less information about some segments than the previous. This may happen because local receive queues are allowed to discard received segments and blocks if they need to reclaim space for other segments with lower sequence numbers (see Section 3.5.6). This particular scenario only occurs because nodes typically send more encoded blocks than the number of blocks they register as being transmitted in their remote receive queues - this is an optimistic strategy that takes advantage of the property of the LT Codes in which only a subset of the transmitted blocks is necessary to decode the segment.

A received acknowledgment may be a duplicate of a previously received one, or may be one that arrived out of order. For this reason, the source node needs to identify these duplicates and old acknowledgments and discard them. This, however, is not achieved perfectly since it is possible for a destination node to discard segments/blocks in its local

receive queue and transmit an acknowledgment with less information than the previous one. Nevertheless, the status information about the segment with lowest sequence number being transmitted never rolls back and thus the source node takes advantage of that to detect received acknowledgments that are surely older than the expected one.

Handshake messages

Handshake messages are used when a session between two nodes is being established. The node that initiates the sessions is referred to as the *initiator node*, whereas the node that responds to the session initiation is referred to as the *respondent node*.

Figure 4.8 shows the structure of an *INIT*, an *INIT_RESP* and a *RESP* handshake message inside a packet. In each of the messages, a nonce is a "one-time" value used to prevent replay attacks. Message *RESP* has two additional fields which are only considered when flag `RFSH` is set. These two fields represent control sequence numbers used to synchronize the initiator and respondent queues when an established session expires and needs to be refreshed (typically because of a segment sequence number overflow).

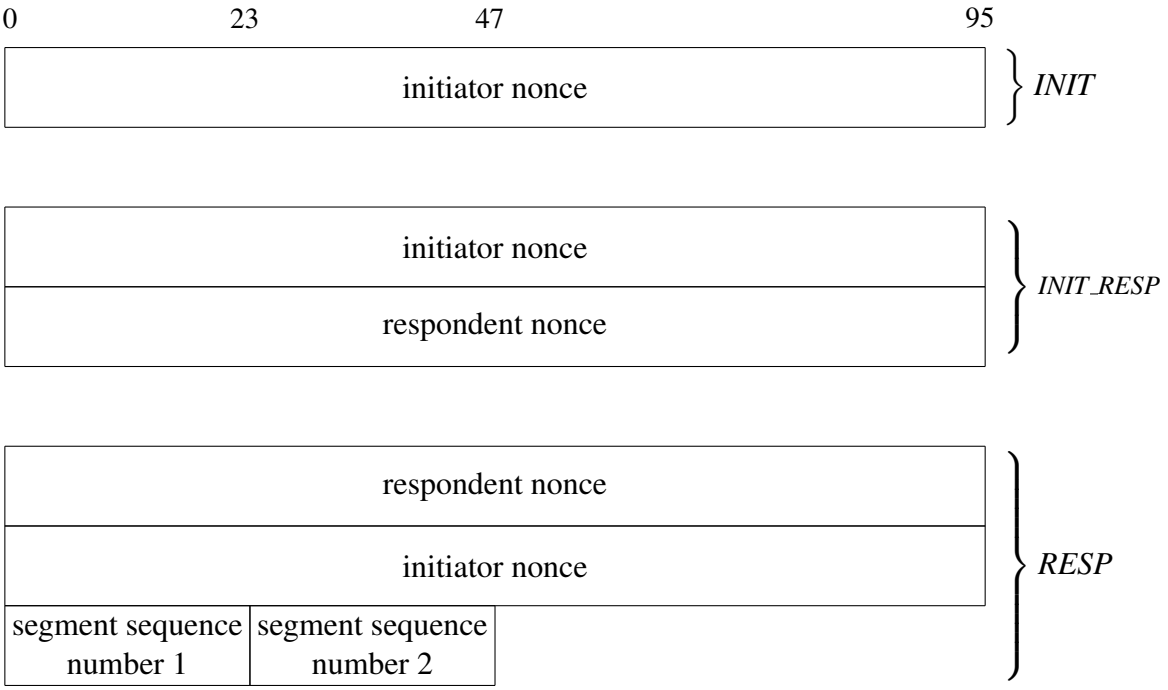

Figure 4.8: Structure of handshake messages inside a packet.

Handshake nonces are constructed in REB by appending a random integer to a timestamp with microsecond resolution. Figure 4.9 shows the structure of a nonce inside a packet.

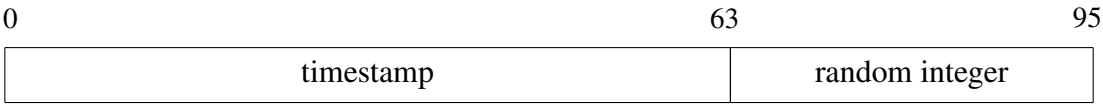

Figure 4.9: Structure of a nonce inside a packet.

4.3.3 Class Diagram

The implementation of the REB library contains multiple classes organized by different packages. Figure 4.10 shows a class diagram with the most important classes from the main package `core.reb`, and some other significant classes from different packages. Class `REB` provides the immediate interface to the application, whereas the remaining classes in the diagram are only used internally and are not directly accessed by the application.

Notable classes are the several classes of the form {⋆}`Task`, whose instances represent actions performed by the internal threads. Classes `ReceiverTask` and `SenderTask` only have one instance each throughout the execution of a REB node and run cyclically since their respective threads (Receiver and Sender threads) must actively wait for events coming from the network (receiver) or other threads (sender). In contrast, classes of the form `MainDispatcher`{⋆}`Task` have multiple instances which are each one constructed and run once every time the dispatcher thread must perform some action triggered by an event coming from the application, the receiver thread or a timeout. Class `REBState` only has one instance that holds the current state of the REB node, and which contains most of the logic for the execution of the dispatcher thread.

Other notable classes are the classes of the form {⋆}`Queue` and the class `SessionManager` which have multiple instances, each one specific to a particular configured remote node. Class `SessionManager` handles the current session between the local node and the respective remote node. Notice the usage of the class `Authenticator` there, where two distinct instances are used by the manager, a *session authenticator* and a *master authenticator*. The former is used to calculate and append a MAC to a sendable data packet (data segment/block or an ACK) - the MAC directed to the destination node. This authenticator depends on the current established session and uses the session key to generate and verify the MAC. The latter authenticator is used to calculate and append a second MAC to a sendable data packet - which is directed to the intermediary node - or to append a MAC to a sendable packet with a handshake message. This authenticator does not depend on any session and uses the original shared secret key configured at start-up (this means that a node is always able to send packets to an intermediary, despite not having an established session with it).

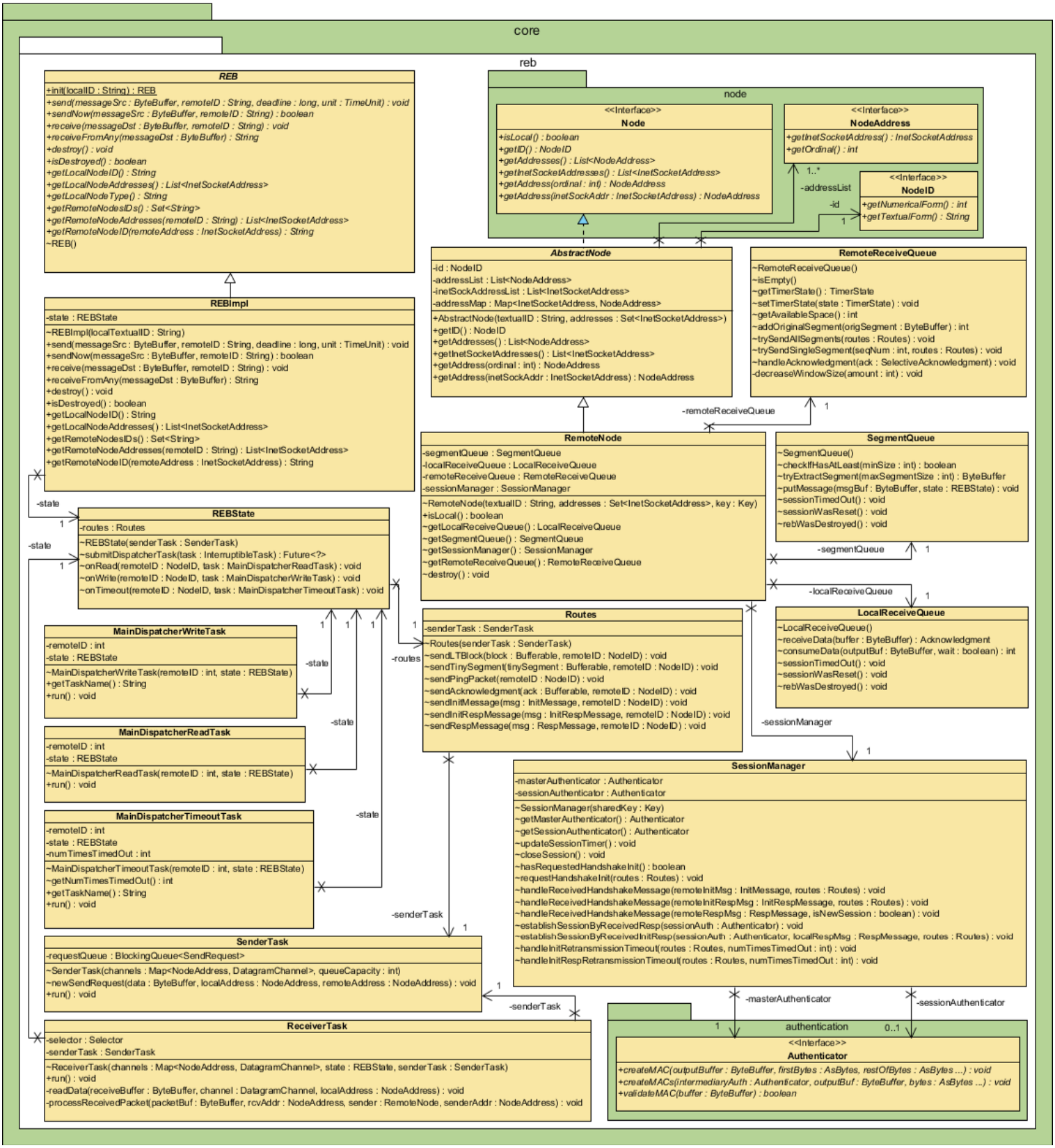

Figure 4.10: Class diagram.

4.3.4 Sequence diagrams

In this section we present two sequence diagrams that illustrate the sequence of events for a sending operation and a receiving operation.

Figure 4.11 shows a sending operation initiated by the application. Here, the application thread selects the appropriate remote node object from a `RemoteNodesMap` instance in order to access the respective segment queue. Then, according to how many bytes are currently in the segment queue, the application thread loops until the whole message is copied into the queue (blocking when it has to wait for space). After completing the copy of the message to the queue, the application thread notifies the dispatcher thread of the new data. This prompts the dispatcher thread to inspect the current state of the communication with the remote node and take the appropriate action according to how many bytes are available it its remote receive queue. Furthermore, if there is no established session by the time the dispatcher thread inspects the queues, then a new one is established before any messages are retrieved from the segment queue.

Figure 4.12 shows a receiving operation initiated by the arrival of a packet from the network. All the actions depicted in this diagram are executed by the receiver thread. First, the thread confirms the validity of the MAC inside the packet, according to the current established session with the remote source node. If the MAC matches, then the packet header is processed. Here, depending on whether or not the packet is directed to the local node, the thread takes the appropriate action. If the packet is directed to a different remote node, then the receiver thread simply forwards the packet to the `SenderTask` object where the sender thread will eventually transmit the packet to its destination. Otherwise, the packet contents are processed. Here, if the header indicates an ACK or a handshake message, the receiver thread simply hands over the packet to the dispatcher thread, where the appropriate action will be taken. If the packet contains a data segment/block, then the receiver thread handles the received packet in the local receive queue.

4.4 Evaluation

Tests were executed on the REB prototype to obtain some relevant metrics about the communication between a source and a destination node. Two overlay network topologies were considered: one in which we configured only one direct path between the source and the destination; another where two additional intermediary nodes were configured, each with two local addresses, which provided eight additional paths in the communication.

A small scale LAN, built for the demonstration of some of the MASSIF components (including REB), was lent to perform tests on the REB prototype. A distinct machine was set-up for each REB node and all machines were connected using four routers. The routers were set-up to redirect traffic following the shortest path between two machines. Two machines running intermediary REB nodes had two network interfaces each, allow-

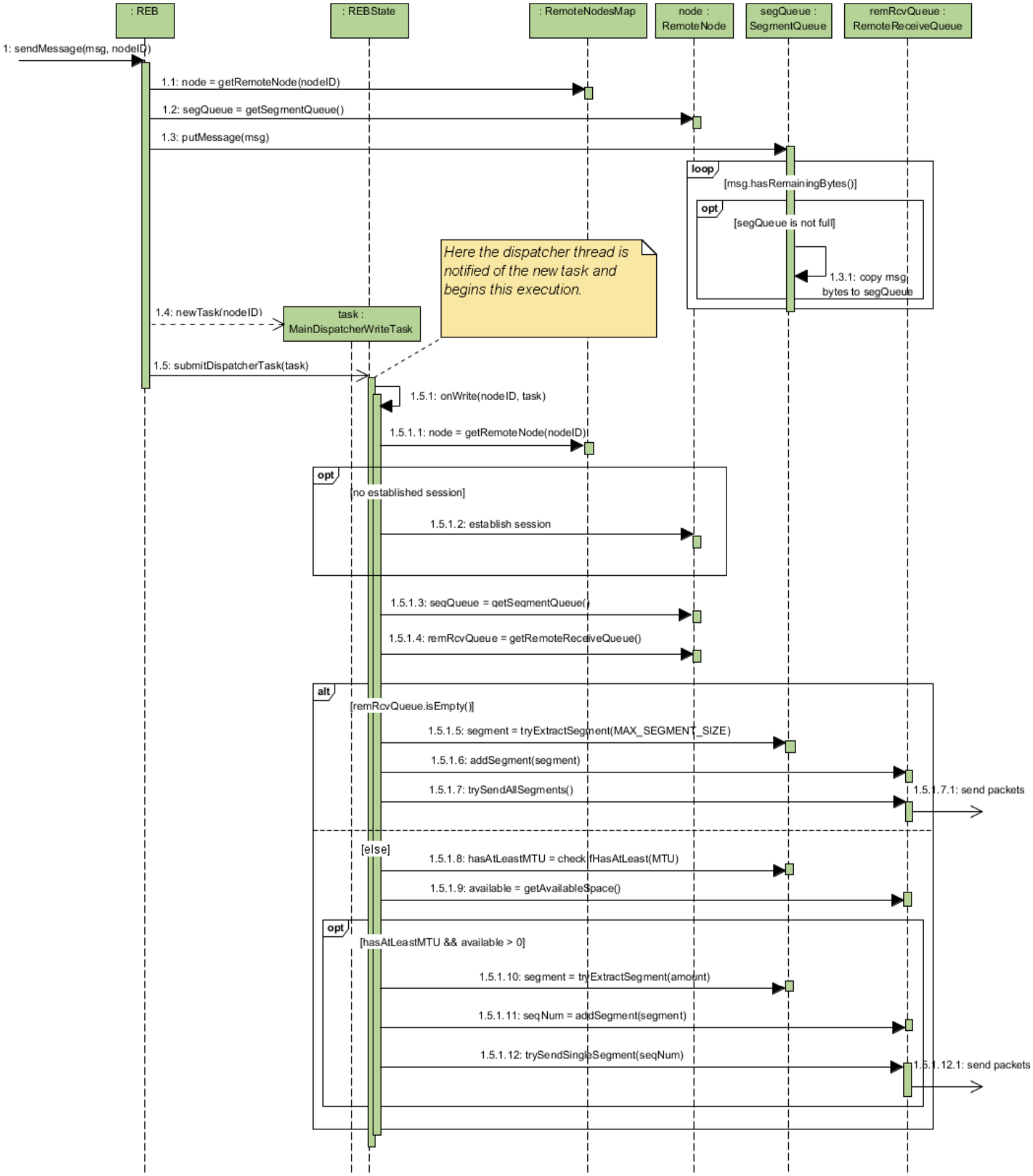

Figure 4.11: Sequence diagram for the sending operation.

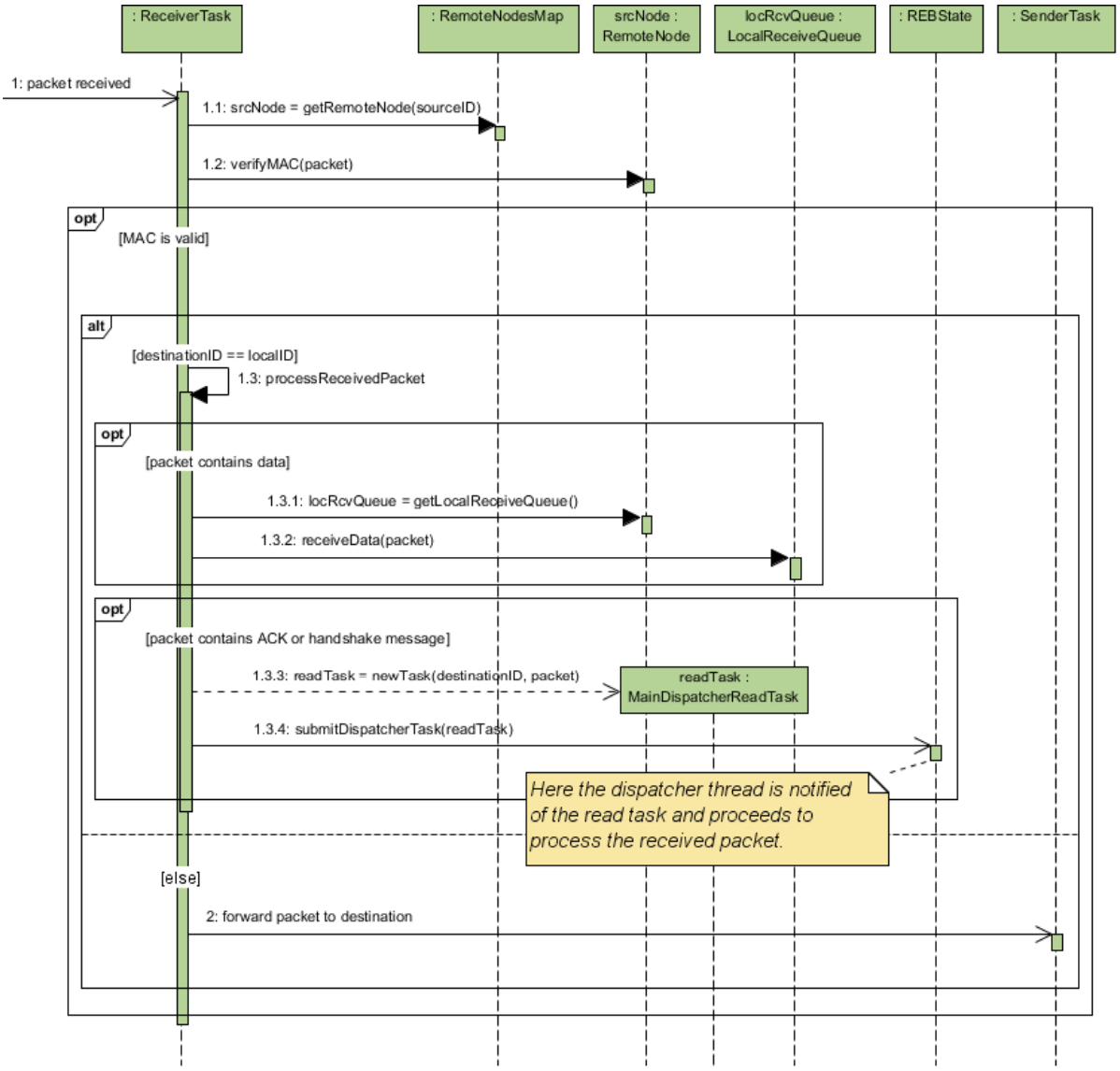

Figure 4.12: Sequence diagram for the receiving operation.

ing for a multihoming configuration on those nodes, which in practice means that each intermediary node could be reached through two different underlay paths by each of the other nodes. Figure 4.13 shows the logical organization of the LAN.

The physical organization of the LAN contained one switch that connected all the machines and routers. Virtual LANs (or VLANs) were used in order to emulate the logical organization described before. Figure 4.14 shows the physical organization of the LAN.

One of the metrics that was obtained from the tests was the average latency of the transmission of a message. For its measurement, the source node transmitted a message to the destination which responded with an acknowledgment message. The source node counted the elapsed time between the moment of sending a message and the moment of

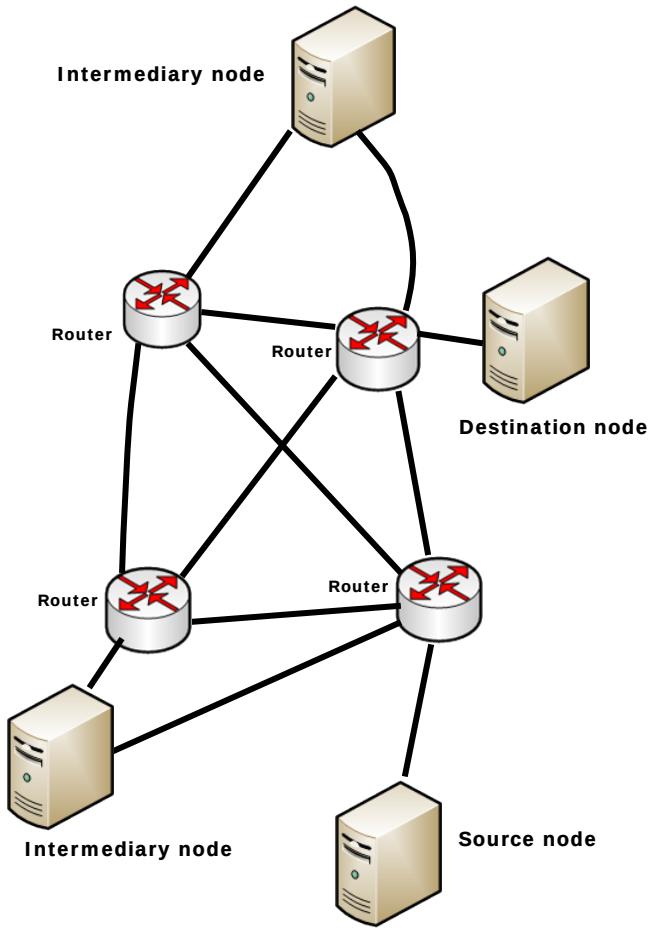

Figure 4.13: Logical organization of the underlay network used for testing purposes.

receiving the acknowledgment, which resulted in the estimation of the Round Trip Time (or RTT) of the transmission. The source node repeated this process a number of times, waiting for the receipt of an acknowledgment before transmitting the next message. In the end, it calculated the average of all stored RTT values and divided the average by two in order to obtain the estimated average latency.

Another obtained metric was the average throughput of a continuous stream of messages. For its measurement, the source node transmitted a continuous stream of messages for 30 seconds and in the end sent one final message which triggered a response from the destination node (with an acknowledgment message). The source node counted the total elapsed time between the moment of sending the first message and the moment of receiving the acknowledgment, and counted as well the total number of bytes transmitted. Finally, it calculated the average throughput, dividing the total number of bytes transmitted by the total elapsed time.

For each measurement, we varied the size of the messages from 1 byte to 10^7 bytes (or 10 MB), visiting all powers of ten in-between. The graphs in Figure 4.15 show the results for the scenario with only one direct path between the source and destination. The

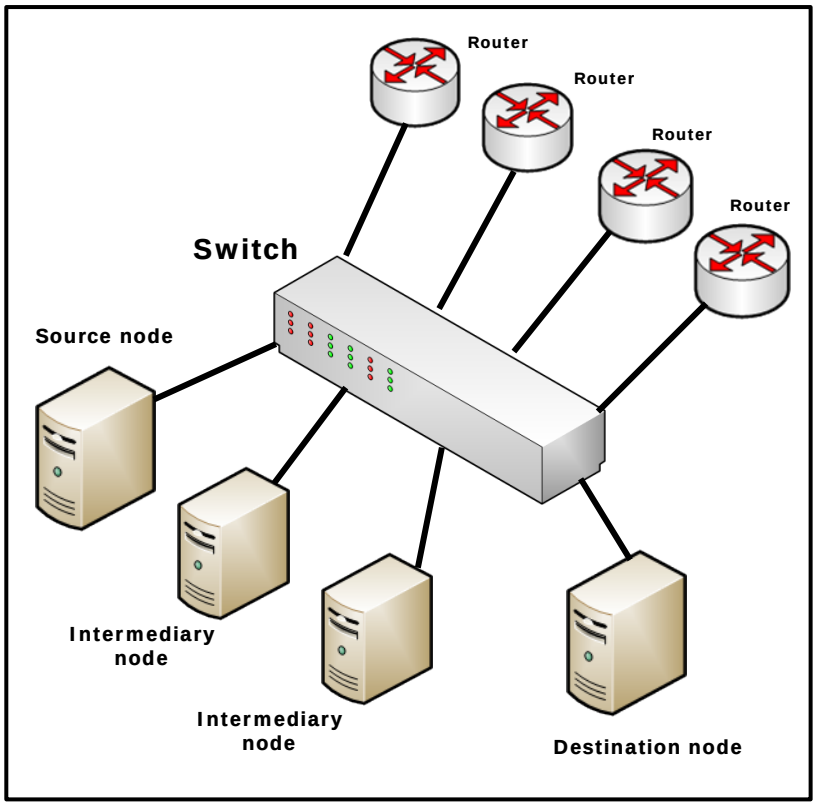

Figure 4.14: Physical organization of the underlay network used for testing purposes.

graphs in Figure 4.16 show the results for the scenario that includes two intermediary nodes. The two vertical lines on each graph indicate landmark message sizes: the one on the left indicates the size of the largest segment which can be completely put inside a REB packet without needing to recur to erasure codes; the one on the right indicates the maximum size of a segment that can be encoded and where each of the encoded blocks have the maximum possible size that can fit inside a REB packet. Note that on the latency graphs, both axis are in a logarithmic scale (both with base 10), while on the throughput graphs, only the horizontal axis is in a logarithmic scale (in base 10).

We can observe on both figures that the average latency is kept relatively small for tiny segments but suddenly jumps when segments become encoded. This increase happens because encoding and decoding a segment adds to the latency of the transmission, but also because the erasure codes (LT codes) always divide a segment in a fixed number of blocks, independently of the segment size. To compensate for this poor efficiency of the LT codes when they encode segments with small sizes, we put a few blocks inside each packet, which decreases the total number of transmitted packets per segment. As the size of a segment grows, the number of blocks we can put inside a packet decreases (because of lack of space), therefore the total number of transmitted packets per segment approaches the actual number of encoded blocks. In the graphs we can observe the results of this optimization and the respective decrease of its efficiency: the increase in latency

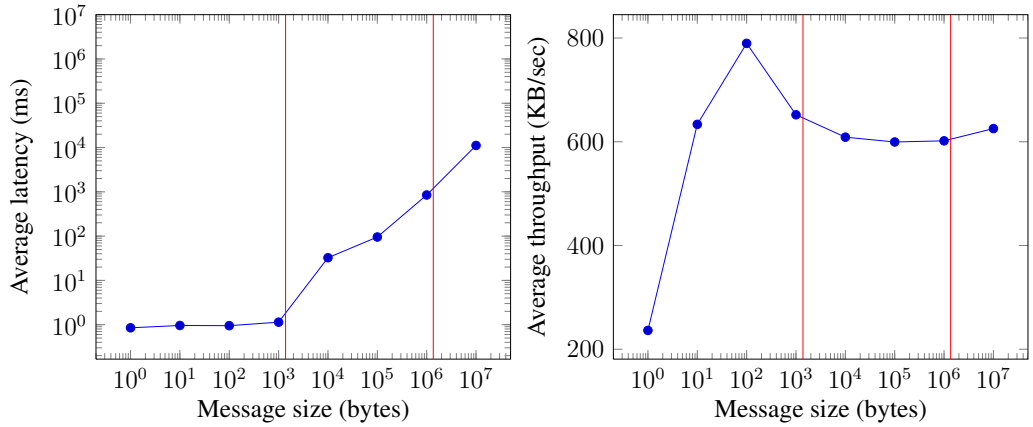

Figure 4.15: Measurements taken from a topology with one direct path.

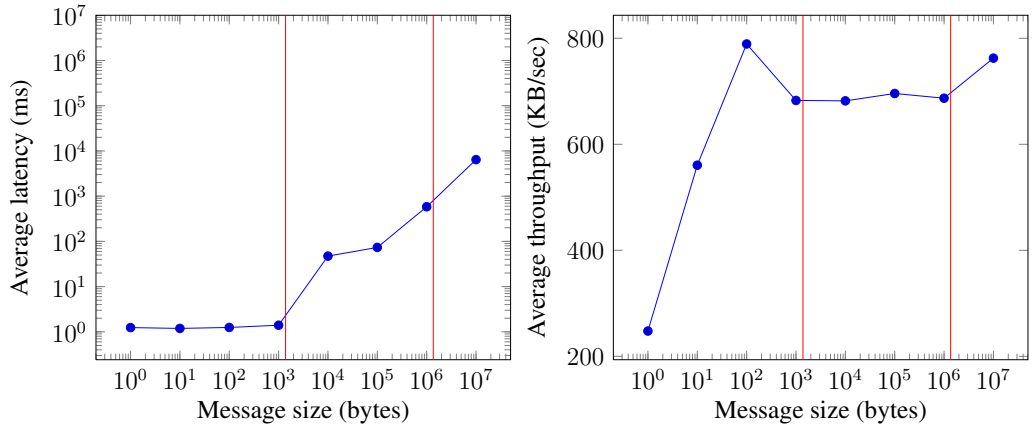

Figure 4.16: Measurements taken from a topology with one direct path and two other two-hop paths.

between messages with sizes 10^5 and 10^6 bytes, and between 10^6 and 10^7 bytes, is more pronounced than the increase between 10^4 and 10^5 bytes.

In both figures, the average throughput graphs show an increase in throughput until a message size of 10^2 bytes, followed by a significant drop at 10^3 bytes. A possible explanation for this is the effect of the flow control which is activated since the source node is trying to send messages faster than the destination can process. In Figure 4.15, there is a second decrease in throughput until a message size of 10^6 bytes. This happens most likely because segments are now being encoded in multiple blocks, producing more packets which consume more bandwidth at the routers, decreasing the overall throughput. Notice, however, how the average throughput increases after the second landmark is reached. Most likely, this happens because encoded segments are now almost always max sized, therefore producing encoded blocks with sizes near the MTU and in this way utilizing the available bandwidth more efficiently. Furthermore, it can be observed that multipath transmission (see Figure 4.16) also increases the average throughput when compared to a single direct path transmission. With more distinct channels there is more overall avail-

able bandwidth so it makes sense that the throughput should increase in this manner.

Chapter 5

Conclusions

Employing a system for intrusion detection is only useful if that system does not fail and cannot be easily attacked and disabled. SIEM tools, despite being excellent at event analysis for detecting intrusions and other system anomalies, are not usually well protected against attacks that might try to compromise the security of their communication. In this thesis we proposed the design of a solution to secure the communication of a SIEM tool, called the REB, which is based on an overlay network, offering a robust and timely communication among its nodes.

The REB applies Message Authentication Codes to the exchanged messages in order to provide node authentication and message integrity. It also utilizes one-hop source-based multipath transmissions, combined with multihoming techniques, in order to tolerate faults in the underlay network and intermediary node crashes, and applies a probing mechanism to choose the best paths for a timely delivery of messages – the probing mechanism is source-based in order to tolerate intrusions in intermediary nodes and avoiding malicious interference in the path selection. Erasure codes are used to minimize redundancy in the transmissions while maintaining the fault tolerance properties of a multipath communication.

Nevertheless, there is still room for improvement and progress in REB will continue. We now present some of the proposed future changes in the design of the REB:

Scalable configuration of the overlay network Instead of a static configuration in which every REB node requires the knowledge about the whole overlay network, an alternative solution must be devised for the system to scale well with the number of nodes involved. A possible idea is to use a centralized system that acts as a network administrator and communicates the entry of new nodes to only some of the existing nodes in the network.

Raptor codes Instead of using LT Codes as an implementation of the fountain codes, we plan to use the more sophisticated and faster Raptor Codes, namely the latest version of RaptorQ [13].

Key exchange mechanism As of now, the REB nodes establish session keys using secret symmetric keys shared between pairs of nodes. These static keys must be configured by hand, which places a burden in the set-up of the nodes. We plan to add a key exchange mechanism that permits a dynamic generation of new keys when necessary. A solution could be to use a pair of public/private keys per node and use the Authenticated Diffie-Hellman mechanism [24] to exchange a new generated shared key per node pair session.

Congestion control Despite the fact that REB employs a flow control mechanism to limit the transmission rate between two nodes, there is no mechanism to avoid congestion at the network level. We plan to add such a mechanism, providing a "TCP friendly" communication that uses, in a fair way, the available bandwidth in links that share the traffic with other TCP connections.

Multicast transmission In some cases, it is beneficial for a SIEM tool to utilize multiple replicas of a correlation engine, in order to tolerate faults or intrusions on individual engine machines. In such a scenario, SIEM sensors may need to transmit multiple copies of events to different engines. To minimize message overhead, REB could provide a multicast communication in which one of two things could happen: either intermediary nodes would help replicating received packets to their destinations, thus removing the need for duplicate transmissions at the source; or source nodes (associated to sensors) would only need to transmit a subset of the encoded blocks that form an event to each destination node (associated to engine replicas), followed by a coordination among the destination nodes to reconstruct the original event.

Acronyms

SIEM: Security Information and Event Management

MASSIF: MAnagement of Security information and events in Service Infrastructures

REB: Resilient Event Bus

IP: Internet Protocol

UDP: User Datagram Protocol

TCP: Transmission Control Protocol

SSL: Secure Sockets Layer

TLS: Transport Layer Security

MAC: Message Authentication Code

ACK: Acknowledgment

SACK: Selective Acknowledgment

RTT: Round Trip Time

MTU: Maximum Transmission Unit

FIFO: First In First Out

JVM: Java Virtual Machine

Bibliography

[1] MASSIF Homepage | MASSIF FP7 Project. http://www.massif-project.eu/.

[2] A. Akella, J. Pang, B. Maggs, S. Seshan, and A. Shaikh. A Comparison of Overlay Routing and Multihoming Route Control. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 93–106, 2004.

[3] Y. Amir, C. Danilov, S. Goose, D. Hedqvist, and A. Terzis. An Overlay Architecture for High Quality VoIP Streams. *IEEE Transactions on Multimedia*, pages 1250–1262, 2006.

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 131–145, 2001.

[5] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the Conference on USENIX Security Symposium*, pages 1–14, 2003.

[6] W. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. *IEEE/ACM Transactions on Networking*, pages 767–780, 2004.

[7] T. Dierks and C. Allen. The TLS Protocol Version 1.0 (RFC 2246). IETF Request For Comments, 1999.

[8] K. Gummadi, H. Madhyastha, S. Gribble, K. Levy, and D. Wetherall. Improving the Reliability of Internet Paths with One-hop Source Routing. In *Proceedings of the Symposium on Operating Systems Design & Implementation*, pages 13–13, 2004.

[9] F. Guo, J. Chen, W. Li, and T. Chiueh. Experiences in Building a Multihoming Load Balancing System. In *Proceedings of IEEE INFOCOM*, pages 1241–1251, 2004.

[10] A. Lane. Securosis Blog | Understanding and Selecting SIEM/LM: Use Cases, Part 1, Securosis Blog. https://securosis.com/blog/

understanding-and-selecting-siem-lm-use-cases-part-1, April 2010.

[11] A. Lane. Securosis Blog | Understanding and Selecting SIEM/LM: Use Cases, Part 2, Securosis Blog. https://securosis.com/blog/understanding-and-selecting-siem-lm-use-cases-part-2, May 2010.

[12] M. Luby. LT Codes. In *Proceedings of the IEEE Symposium on the Foundations of Computer Science*, pages 271–280, 2002.

[13] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder. RaptorQ Forward Error Correction Scheme for Object Delivery. IETF RFC 6330, 2011.

[14] D. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, 2002.

[15] MASSIF Consortium. *Deliverable D2.1.1 - Scenario requirements*. Project MASSIF EC FP7-257475, April 2011.

[16] MASSIF Consortium. *Deliverable D5.1.1 - Preliminary Resilient Framework Architecture*. Project MASSIF EC FP7-257475, September 2011.

[17] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. IETF RFC 2018, 1996.

[18] D. Miller, S. Harris, A. Harper, S. VanDyke, and C. Blask. *Security Information and Event Management (SIEM) Implementation*. McGraw-Hill Osborne, November 2010.

[19] R. Obelheiro and J. Fraga. A Lightweight Intrusion-Tolerant Overlay Network. In *Proceedings of the IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 496–503, 2006.

[20] R. Oppliger. Security at the Internet Layer. *IEEE Computer*, pages 43–47, 1998.

[21] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. IETF RFC 6298, 2011.

[22] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, pages 2551–2567, 2006.

[23] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 160–173, 2001.

[24] W. Stallings. *Cryptography and Network Security: Principles and Practice*. The William Stallings Books on Computer and Data Communications. Pearson/Prentice Hall, 2006.

[25] A. Williams. The Future of SIEM - The market will begin to diverge, Amrit Williams Blog. http://techbuddha.wordpress.com/2007/01/01/the-future-of-siem-%E2%80%93-the-market-will-begin-to-diverge/, January 2007.