ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Diego Adolf

Design and Implementation of SAFT on ANA

Semester Thesis, SA-2008-12
March 2008 until August 2008

Advisors: Ariane Keller, Simon Heimlicher
Supervisor: Prof. Dr. Bernhard Plattner

Contents

1 Introduction **1**
1.1 Main Goals 2

2 Background Information **3**
2.1 ANA: Autonomic Network Architecture 3
2.1.1 ANA Node 3
2.1.2 Bricks 4
2.1.3 Compartments 4
2.1.4 Information Channels and Information Dispatch Points 5
2.1.5 Primitives 5
2.2 SAFT: Store And Forward Transport 6
2.2.1 Overview 6
2.2.2 End-to-End Sublayer 7
2.2.3 Hop-by-Hop Sublayer 7
2.3 IP Compartment 8
2.3.1 Next Hop Address 9
2.3.2 Node Communication 9

3 Design of SAFT for ANA **11**
3.1 Overview 11
3.2 End-to-End sublayer 11
3.2.1 End-to-End Main Brick 12
3.2.2 Connection Congestion Control Brick 14
3.2.3 Connection Flow Control Brick 15
3.3 Sublayer Interface Brick 15
3.4 Hop-by-Hop sublayer 16
3.4.1 Hop-by-Hop Main Brick 16
3.4.2 Link Congestion Control Brick 17
3.4.3 Link Flow Control Brick 19
3.5 Comparison to legacy network implementation 19

4 Implementation **20**
4.1 Overview: Current State 20
4.2 Compartment Specific Settings 20
4.2.1 SAFT Header 21
4.2.2 Segment and Fragment types 24
4.2.3 XRP messages 24
4.2.4 Application MTU 25
4.2.5 Segment, Fragment and Packet Sizes 26

4.3 End-to-End sublayer . . . 26
4.3.1 End-to-End main Brick . . . 26
4.4 Sublayer Interface Brick . . . 32
4.5 Hob-by-Hop sublayer . . . 36
4.5.1 Hop-by-Hop Main Brick . . . 36
4.5.2 Link Congestion Control Brick . . . 40
4.6 Test Application: File Transfer . . . 44

5 Validation **46**
5.1 Overview . . . 46
5.2 Test 1: Multiple Transmissions . . . 47
5.3 Test 2: Single Transmission . . . 48

6 Conclusion and Future Work **49**
6.1 Conclusion . . . 49
6.2 Future Work . . . 49

Bibliography **51**

Appendix **53**

A Appendix **53**
A.1 Usage . . . 53
A.2 Doxygen Documentation . . . 53
A.2.1 SAFT End-to-End Main Brick . . . 53
A.2.2 SAFT Sublayer Interface . . . 63
A.2.3 SAFT Hop-by-Hop Main Brick . . . 73
A.2.4 SAFT Link Congestion Control Brick . . . 84
A.2.5 SAFT Main Include File . . . 99
A.2.6 SAFT Demo Brick . . . 118
A.3 Assignment . . . 129

Abstract

The main motivation for this thesis is to provide the ANA prototype with a transport protocol. Since the primary goal of ANA is versatility and not compatibility, we looked for a modular and highly adaptive transport protocol instead of implementing TCP. We chose SAFT, as it is designed for challenging scenarios and is modular.

By splitting the SAFT functionalities into independent modules, so-called bricks, we also aim at providing the groundwork for other transport protocol implementations. With some modifications to the current design and implementation, basic versions of UDP or TCP can be created.

The current implementation, even though only incorporating a fraction of the SAFT functionalities, provides a useful service for applications as shown in our validation. Applications can resolve another node through the SAFT compartment and send data to it using a semi-reliable communication channel.

1 Introduction

Today communication amongst most computers connected to the Internet or local area networks is handled using the Internet Protocol Suite specified in [RFC1122] dating back to 1989. This suite, also known as TCP/IP because of its most prominent protocols, has a layered structure with IP as the core protocol similar to that of the OSI reference model [OSIRM]. While this structure allows flexible configuration, it strictly requires every node to run IP. For some target scenarios, such as (mobile) ad hoc networks, sensor networks or peer-to-peer networks, IP may be inappropriate [ANAB, p. 2].

To deal with this problem, is one of the objectives of the **ANA** project **(Autonomic Network Architecture)**. It aims at creating an architectural framework which allows coexistence and communication between different types of networks using a "minimum generic interface" [ANAB, p. 2]. The new architecture should allow networks to perform functional scaling both horizontally (more functionality) as well as vertically (different ways of integrating functionality); the ultimate goal being: " [...] a novel autonomic network architecture that enables flexible, dynamic, and fully autonomous formation of network nodes as well as whole networks." [ANAP].

The ANA project also aims at providing an implementation to test and demonstrate the concepts developed. Currently a few basic protocols, e.g., Ethernet and IP, have already been adapted to run on the ANA prototype. However, it is not one of the objectives of the project to solely reproduce traditional protocol stacks. It allows for new protocols, such questioning existing principles like the end-to-end approach, to be developed and used where they provide advantages compared to traditional ones. This is the case with hop-by-hop transport protocols that clearly outperform standard end-to-end transport protocols like TCP in wireless mobile networks.

As part of the Internet Protocol Suite, TCP has become the most widely used transport protocol. It was originally designed for wired networks with a stable topology [DPD]. In such networks most packet loss occurs because of congestion at certain nodes [CAC]. TCP, as an end-to-end protocol, has congestion control mechanism implemented at the source and the destination node. In wireless mobile environments however, this approach is shown not to be optimal [HBT]. This is because in wireless mobile networks packet loss is mainly due to route failure and link errors and not due to congestion.

To address this matter, the authors of [TLR] conclude that " [...] it seems helpful to include the intermediate hosts in the data transfer". Protocols that implement intelligence in intermediate nodes are called hop-by-hop protocols. The simulation experiments done in [TLR] show that their hop-by-hop protocol achieves up to three times faster delivery of messages than

TCP in mobile networks.

In this thesis we design and implement the first transport protocol for ANA: **SAFT (Store And Forward Transport)**, a hop-by-hop transport protocol specially suited for wireless mobile networks.

Hop-by-hop protocols illustrate the fact that new technologies make it necessary to reconsider current standards like the Internet Protocol Suite and search for new ways of communication in order to cope with ongoing changes.

1.1 Main Goals

As a full implementation of SAFT was outside the scope of a semester thesis, we proposed ourselves three main goals that were to be reached at the end of our work:

- Provide the basic functionalities for a reliable data transport solution, specially on wireless mobile networks
- Achieve an implementation that complies with ANA guidelines
- Validate the implementation through test cases

2 Background Information

In this chapter we provide the reader with the concepts that are essential for the understanding of this thesis. Our work is mainly founded on the ANA core architecture and the SAFT design described in [TLR] and [SAFT]. Also, some information on the IP compartment for ANA is provided, as it is the network compartment that the current implementation uses.

2.1 ANA: Autonomic Network Architecture

The ANA project's main documentation consists of [ANAB] and [ANAC]. Adhering to these, we will explain three concepts that we consider important: the typical setup of an ANA node, the bricks that provide functionalities and the compartment concept.

2.1.1 ANA Node

A typical ANA node is divided into two main parts as shown in fig. 1: the Minmex and the ANA Playground.

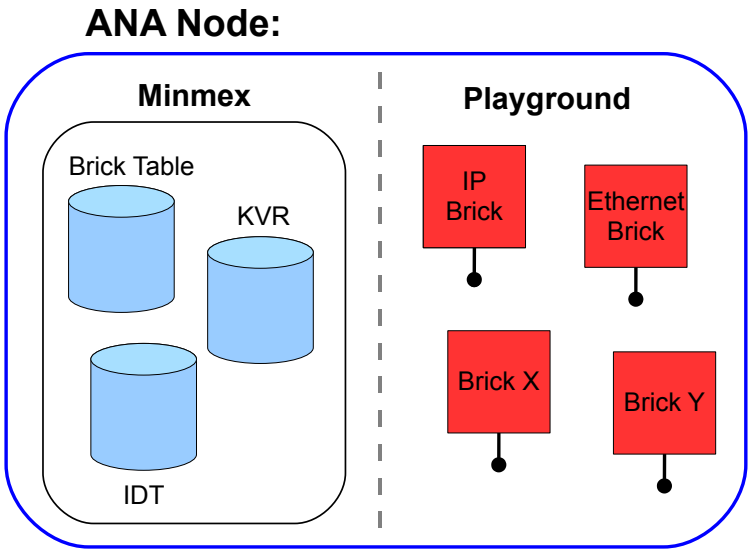

Figure 1: Exemplary ANA node

The Playground holds all software modules providing (network) functionalities for ANA. These modules, so-called bricks, are explained in section 2.1.2.

The Minmex is needed for those modules to communicate with each other. It is an essential part of ANA and is present in every node.

2.1.2 Bricks

Bricks can be thought of as software modules. They are present in the ANA playground and are the units that actually provide functionality to the node. In ANA most implementations of network protocols (or other network related software) split the functionalities of their software into several bricks. Thus, an implementation normally consists of two or more interdependent bricks which as a group provide the targeted functionality. This group of bricks is often called a compartment, even though the term has a vaster meaning (see 2.1.3). The brick representing this compartment within the ANA Playground is called compartment provider brick. It is the gateway for other bricks wanting to use the functionalities offered by that compartment.

One of the advantages of dividing functionalities into several bricks is that a single brick providing a very specific functionality, can easily be reused by other applications. A good example for this is the checksum brick of the IP compartment which offers a functionality useful for other compartments as well.

2.1.3 Compartments

The official definition of a compartment states the following: "A compartment is defined by the set of abstract entities (members) which are able and willing to communicate among each other according to compartment's operational and policy rules." [ANAB, p. 8]. This very flexible definition allows the use of compartments in many different scenarios and is regarded as one of ANA's most important concepts. Because of the complexity of the compartment concept, we will mainly focus on it's applications within SAFT for ANA. (For a complete description see [ANAB, sec. 3.1].)

The most relevant uses of compartments for this thesis are related to the IP compartment. The IP compartment groups all nodes connected to each other running the corresponding IP bricks (see fig. 2). All nodes that are members of that compartment then can communicate with each other using the IP protocol. Important to mention is that within the IP compartment a set of nodes, e.g., a subnet, is represented as a compartment as well. This implies that compartments can be overlaying or represent a subset of another compartment.

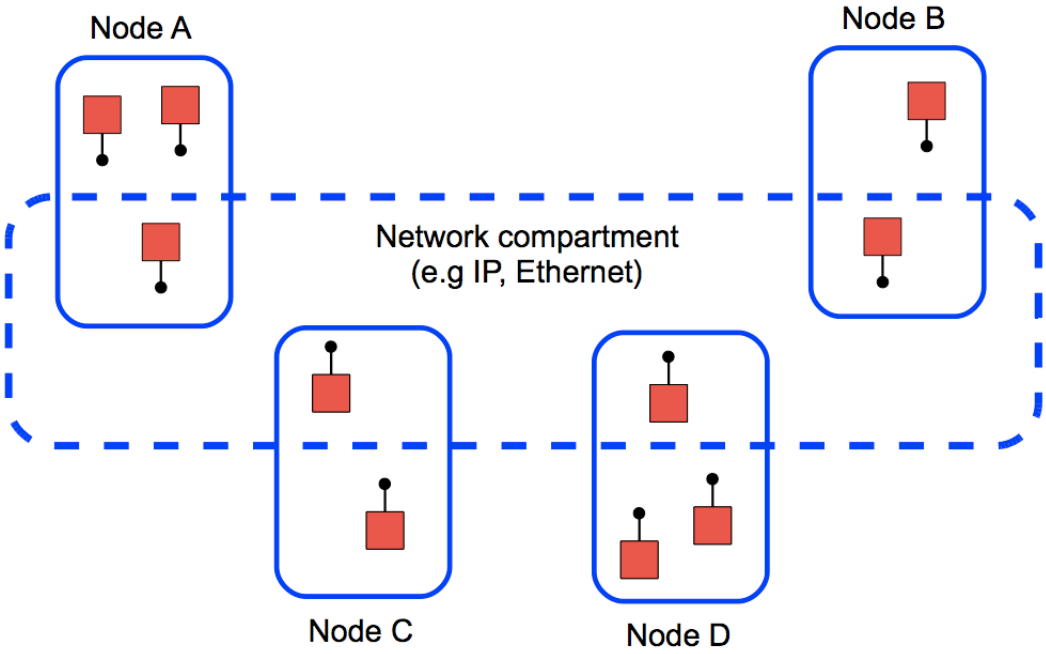

Figure 2: Several nodes as members of a network compartment

2.1.4 Information Channels and Information Dispatch Points

Information channels (IC's) abstract the communication service provided by a compartment. They are used to establish a communication channel between two bricks. See section 2.3.2 for their use within the IP compartment.

Information dispatch points or IDP's, are gateways for accessing bricks or information channels. A brick or a compartment can create several IDP's in order for other bricks to access its functionalities. In most graphics they are represented as black dots at the entrance points of bricks and information channels.

2.1.5 Primitives

All communication among bricks in the ANA Playground is done using *primitives*. The basic primitives implement fundamental communication paradigms which need to be supported by every compartment. This is necessary to provide a generic way for all compartments to interact with each other, regardless of the functionality they provide. The five basic primitives are:

- *publish*

With this primitive we can publish a certain service in a compartment and make that service reachable through that compartment. This is necessary if, e.g., we expect data that will arrive through that compartment. After the publish command, that compartment can locate us and forward the data to us.

- *unpublish*
  This command reverts the publish command, i.e., a service previously published in that compartment will not be available anymore through that compartment.

- *resolve*
  Using the resolve primitive we can request a communication channel through a certain compartment to a service previously published in that compartment.

- *lookup*
  This primitive requests information from a compartment on a certain service published inside that compartment.

- *send*
  With this primitive we can send data to a service we have resolved before using the resolve primitive

The above explanations have been simplified. For a in depth description of ANA primitives see [ANAB, sec. 3.3.3].

2.2 SAFT: Store And Forward Transport

Our design and implementation of SAFT for ANA are based on two publications: [TLR] and [SAFT]. The following paragraphs summarize the structure and functionalities of SAFT.

2.2.1 Overview

As already mention in the introduction, SAFT is a hop-by-hop transport protocol. Its main structure consists of two sublayers: the end-to-end sublayer and the hop-by-hop sublayer. The separation into these two sublayers allows the protocol to perform control mechanism separately for the connection and for the links involved in the connection.

A *connection* represents the data connection between the source and the destination node and is controlled by the end-to-end sublayer. The hop-by-hop sublayer controls each *link*, which refers to a data connection between two neighboring nodes on the route of the global connection. Figure 3 shows a four node setup illustrating this. Here the network has been abstracted for the sake of simplicity.

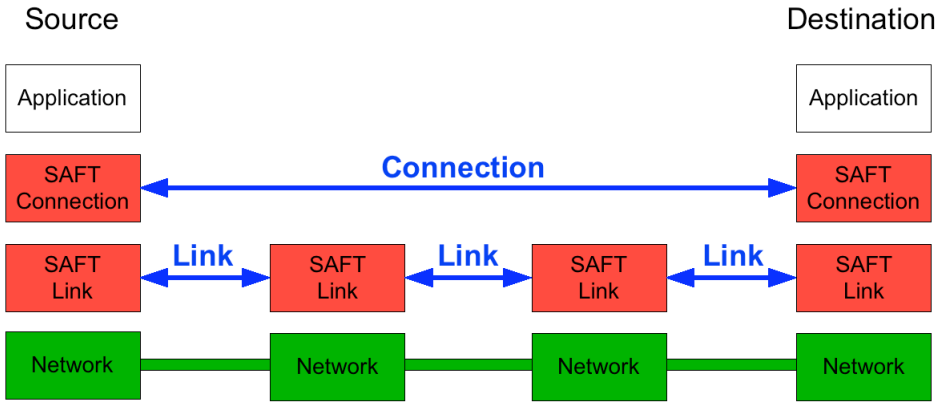

Figure 3: Illustration of terms *connection* and *link* on a multi-hop connection

2.2.2 End-to-End Sublayer

The end-to-end sublayer handles the transmission requests from applications. It is only active on the source and the destination node. The destination node's only tasks are to receive and reorder the data and acknowledge it to the source. The source node is solely responsible for performing congestion and flow control on the connection.

Similar to regular layers known from the OSI reference model [OSIRM], this sublayer implements its own data unit, namely segments. The application requesting a connection passes the data to the end-to-end sublayer which then splits it into segments before further processing. These segments are then passed on to the hop-by-hop sublayer which takes care of the actual data transmission.

SAFT has no connection establishment procedure prior to data transmission like there is with TCP. Data received from the application is directly sent to the destination node.

2.2.3 Hop-by-Hop Sublayer

As opposed to traditional transport protocols, which only run on both ends of a connection, SAFT's hop-by-hop runs on every node. This means that

data is passed to the hop-by-hop sublayer on every intermediate node. Figure 4 shows the data flow for a multi-hop connection illustrating this fact.

From the diagram one can also notice that the hop-by-hop sublayer is the unit actually interacting with the network compartment or network layer respectively for ANA and traditional network architectures. The end-to-end sublayer passes its segments to the hop-by-hop sublayer which again splits these segments into fragments before sending them over the network (e.g., using IP or any other network protocol). At the destination node the received fragments are reassembled and returned to the end-to-end sublayer as segments.

One fact to be recalled is that fragments, even though part of the same segment, may use different routes to reach the destination node. This is very useful in scenarios where link failures occur often as it is the case in wireless mobile networks.

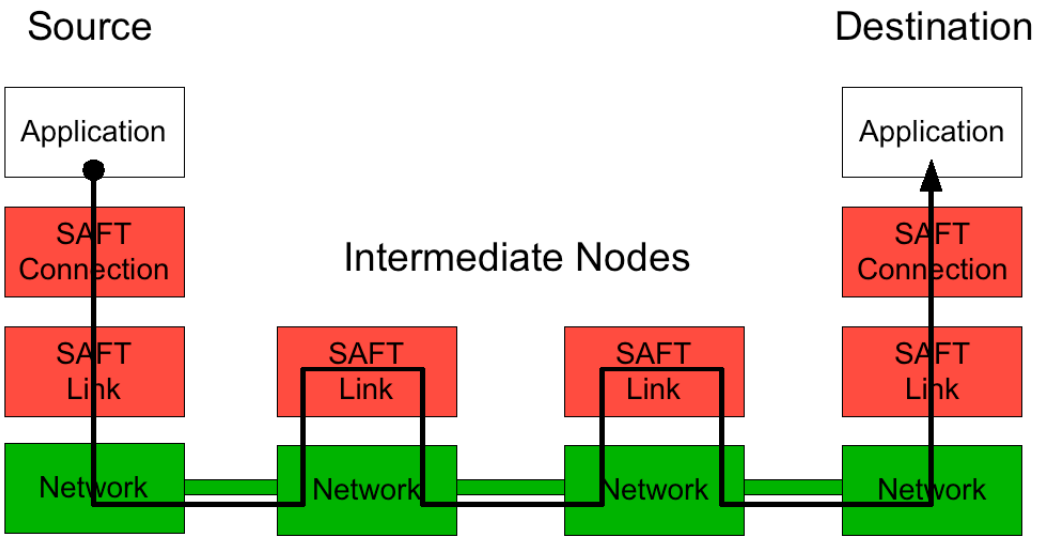

Figure 4: Data flow for a multi-hop connection

2.3 IP Compartment

In the current implementation of SAFT, the IP compartment is the network compartment used. Its task are to provide the address of the next hop and open a communication channel to it. These tasks are explained below. For further details on IP refer to [IPCO].

2.3.1 Next Hop Address

The next hop is the next node to be reached on the route leading towards the destination node from the current node's point of view.

Every time a fragment is to be sent, the hop-by-hop sublayer request the address of the next hop. This is necessary, because every fragment may take a different route to reach the destination node.

Using the IP compartment, this can be done by querying the IP forwarding brick. This brick builds up a forwarding table with the help of a routing protocol. In our case this was done using the RIP brick.

Once the address of the next hop is acquired through the IP forwarding brick, a communication channel to the required node can be requested through the IP compartment's main brick.

2.3.2 Node Communication

In order to send fragments, SAFT requests the IP compartment to create a communication channel by which data can be passed directly to the SAFT instance running on the next hop.

In the ANA world such a communication channel consists of two elements: an IDP (Information Dispatch Point) and an IC (Infomation Channel). The information channel abstracts the transmission's underlaying process. The IDP serves as a gateway for this information channel. The sending node simply deploys the data to be sent at this IDP and the data will be forwarded to the next hop through the information channel. In fig. 5 two nodes connected through an IC are depicted. SAFT's hop-by-hop sublayer running on the source node requests an IC using the network compartment in order to forward the data from the application to the next hop. When the transmitted fragments arrives at the destination node, the data will be returned to the end-to-end sublayer and from there passed on to the corresponding application.

Worth mentioning in this context is that ANA's modular structure allows it to use protocols in a different order as in normal network architectures. For example, one could run SAFT directly over Ethernet, thus skipping the network layer of traditional architectures. The process of connecting two nodes as shown in fig. 5 would still be the same, only addressing would be done using MAC addresses instead of IP addresses.

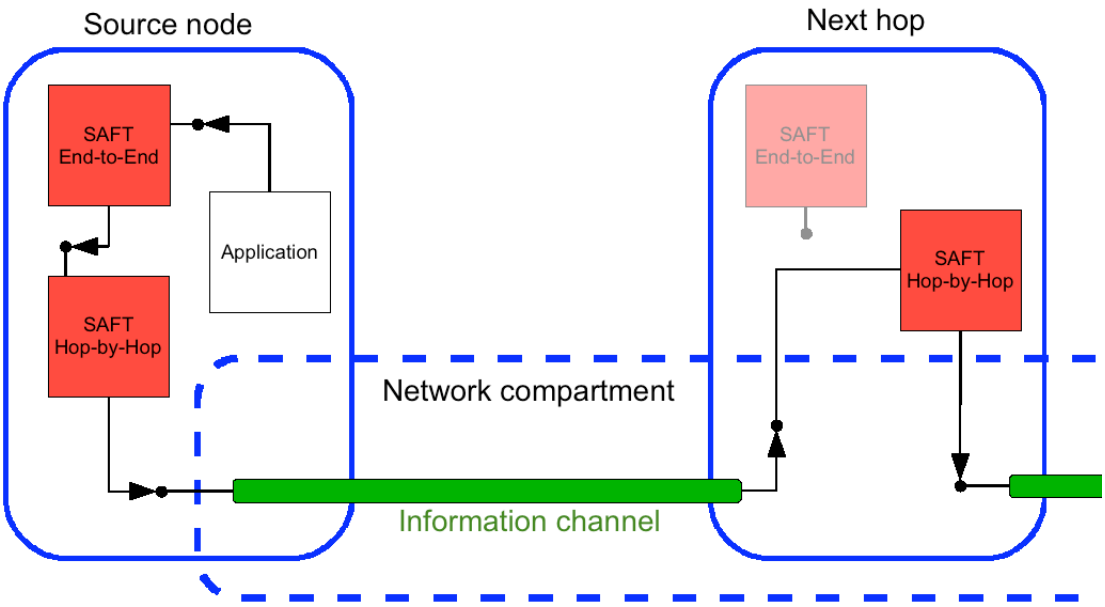

Figure 5: Data transmission to next hop through network compartment

3 Design of SAFT for ANA

For the design of SAFT for ANA we guided ourselves by two main objectives:

- Isolate the main functionalities of SAFT and assign them to bricks
- Make both SAFT sublayers work independently

The purpose of these two objectives is to create a design that makes it easier for other applications to reuse our bricks.

To reach the first objective, we analyzed the framework for hop-by-hop transport protocols proposed in [TLR]. We designed a brick for each of the main functionalities extracted out of this framework and specified its tasks. The second objective was reached by designing a brick that decouples the dependencies of the hop-by-hop and the end-to-end sublayer and acts as an interface when communicating with the sublayers.

The sections below show how the SAFT functionalities have been assigned to different bricks and what the task of each brick are.

3.1 Overview

The skeleton of our design consists of the end-to-end and the hop-by-hop sublayer. Each of these sublayers holds three bricks providing the functionalities of their corresponding sublayer. An additional brick, the so-called sublayer interface, connects these two sublayers by providing the communication mechanisms necessary.

Each sublayer has one main brick: the *SAFT end-to-end* and *SAFT hop-by-hop* brick respectively. The main bricks are the only bricks actively interacting with elements outside the SAFT compartment. The SAFT end-to-end brick is the compartment provider brick and is responsible for processing the requests from applications. The SAFT hop-by-hop brick is the brick interacting with the network compartment.

The control mechanisms of each sublayer are implemented as separate bricks and only communicate with bricks inside their own sublayer. Figure 6 shows the arrangement of bricks inside the SAFT compartment. The arrows shows which bricks interact with each other.

3.2 End-to-End sublayer

The end-to-end sublayer sublayer, is active on the source and the destination node and interacts directly with the application. It is the sublayer responsible

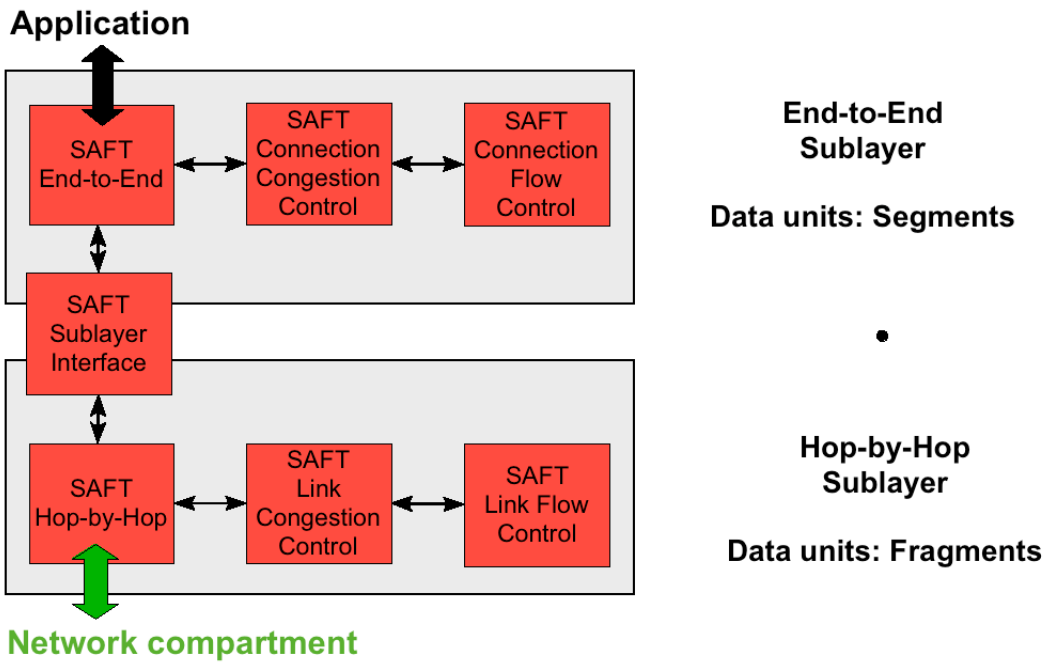

Figure 6: Structure of SAFT design and arrangement of bricks

for guaranteeing end-to-end reliability, i.e., making sure that data arrives consistently at the destination node.

To achieve this, three basic functions are implemented separately in bricks: connection-based send and receive functionality, connection congestion control and connection flow control. Each brick and its tasks are described below in a separate subsection.

3.2.1 End-to-End Main Brick

This brick is responsible for interacting with the applications. It also is the brick that handles the data on the end-to-end sublayer, i.e., takes care of sending and receiving it. Its functions are described in detail below.

Handle Application Request

This brick, being the compartment provider brick, handles the transmission request by applications on the source node. It is also responsible for delivering the transmitted data to the corresponding application upon arrival at the destination node.

Create Segments from Application Data

As soon as an application passes its data to the SAFT compartment for transmission this data is packed into *segments*. These are data units that contain the application data and have a header preceding the data. This header contains all information needed to deliver the data consistently to the destination node and implement control mechanisms. (For implementational details on the header see section 4.)

Sending data
Every segment (representing application data) is stored locally before sending. To send a segment, sending permission has to be requested through the connection congestion control (CCC) brick. If sending permission is granted, the segment is passed on to the sublayer interface which takes care of the rest of the process. In case sending permission is denied, the segment is stored for later use, i.e., it will be resent by order of the CCC brick.

Note: SAFT has no connection establishment procedure, i.e., data is sent directly. Connection establishment is not suitable for wireless mobile networks, because frequent link failures difficult that process.

Note: The segment buffering system (e.g., a dynamically allocated buffer) is managed exclusively by the CCC brick. This means, the end-to-end main brick implements the buffer but only stores newly created segments. Deletion and retransmission of segments is ordered directly by the CCC brick.

Receiving data
When the application data has arrived at the destination node, the sublayer interface passes the data to the end-to-end main brick in form of segments. This brick then delivers the segments to the corresponding application and reorders them previously in case they have arrived out of order (e.g., due to packet loss on the route). For every received segment a *segment acknowledgement* is sent back to the source informing the sending node that the data has arrived at the destination. This also applies if a segment is received more than once (e.g., because of a lost acknowledgement).

Note: Segment acknowledgements are also received by this brick. However, they are not processed locally but are passed on to the CCC brick as soon as they arrive.

3.2.2 Connection Congestion Control Brick

The connection congestion control (CCC) brick implements the congestion control mechanisms of the end-to-end sublayer. The design of this mechanism is explained in the following paragraphs.

Congestion Control
The CCC brick runs only on the source node. It decides if the end-to-end main brick is allowed to send a certain segment or not. It limits the amount of unacknowledged segments for each connection. The end-to-end main brick request sending permission for every segment it wants to send. If the transmission windows for the connection of that segment is not yet exhausted, i.e., the number of unacknowledged segments has not been exceeded, permission is granted. Else, permission is denied and the CCC brick send a transmission order for that segment to the main brick as soon as an older segment is acknowledged.

Connection Management
In order to manage the connections initiated by the end-to-end main brick a system has to implemented by which the status of each connection and segment can be stored and updated. This is necessary to keep count of unacknowledged segments and the status of the transmission window.

Segment Acknowledgements
The end-to-end main brick passes all segment acknowledgements without processing them to the CCC brick. Segment acknowledgements are needed to update the status of segments (i.e., mark them as acknowledged) and their corresponding transmission windows.

Segment Retransmissions
As data can get lost on its way towards the destination node for several reasons, segments may need to be retransmitted. For this to work efficiently, the connection management system should implement timers to detect if a segment has not been acknowledged after its timeout. Retransmission of unacknowledged segments should be implemented respecting exponentially increasing back-off intervals. This guarantees system stability and allows fair coexistence with other protocols [TLR, p. 2].

3.2.3 Connection Flow Control Brick

This brick was not included in the initial design as it seemed that most connection-based control mechanism could be implemented in the connection congestion control brick and that link-based flow control was sufficient. Nonetheless, future implementations should consider flow control for the end-to-end sublayer.

3.3 Sublayer Interface Brick

This brick is the interface between both sublayers and handles the communication between them. Mainly this means converting the data units used by one sublayer into the data units of the other sublayer, i.e., split segments received from the end-to-end sublayer into fragments for the hop-by-hop sublayer and vice versa.

This brick also helps to separate the dependencies between the two sublayers. The idea is to allow independent use of each sublayer. In case the functionality of a single sublayer is reused for a certain implementation, only the sublayer interface brick needs to be adapted. The advantage of this approach is that modifying the sublayer interface is much easier than any other brick because it has a very simple structure and contains only limited functionality. An example for this would be the implementation of and end-to-end protocol needing only the functionalities of the end-to-end sublayer.

This brick performs the following tasks:

Split Segments
Every segment that needs to be sent over the network will be handed over to this brick by the end-to-end main brick. This brick will split the received segments into fragments and then pass them on to the hop-by-hop main brick which takes care of the actual data transmission using a network compartment.

To create the fragments, the application data contained in the segment will be divided into several chunks. The header of the original segment will be appended to these chunks and the fragment number field in the header modified according to the order of the data. (For details on the header see section 4.)

Reassemble Segments
Every fragment received by the hop-by-hop main brick at the destination node will be passed to this brick. This brick will sort out all fragments corre-

sponding to the same segment, extract the application data of each fragment and create a segment containing the original header and the application data. The application data can be restored using the fragment numbers. Note that this brick needs to implement reordering of fragments as they may arrive out of order because retransmissions.

3.4 Hop-by-Hop sublayer

The hop-by-hop sublayer implements the actual *store and forward* mechanism on each link a of connection and thus runs on every node involved in it. The goal is to make data transmission between two neighboring nodes more resilient to link and route failures. This is specially useful in wireless mobile networks were such errors occur often.

This sublayer's functionality is divided into three bricks implementing the following functions: link-based send and received functionality, link congestion control and link flow control. Each brick is described below in a separate subsection.

3.4.1 Hop-by-Hop Main Brick

This brick is the unit handling the application data received from the end-to-end sublayer through the sublayer interface. It is the only brick interacting with the network compartment and thus, is responsible for requesting connections to the next hop and forwarding the application data. The functions of this brick are described in detail below.

Store and Forward Procedure
The store and forward procedure applies to all fragments that need to be forwarded to another node. This means, all fragments arriving either from the end-to-end sublayer or from another node and still on their way to their final destination. These fragments are stored locally and then sent to their corresponding next hop. As soon as a fragment has been acknowledged by the node receiving the data, i.e., the next hop, the fragment can be deleted locally. Otherwise, it is kept for retransmission or is deleted after a certain amount of retransmission attempts.

Sending Data
All data to be sent comes either from the end-to-end sublayer or another node. This data is received as fragments and is stored locally in some sort of buffer system (e.g. a dynamically allocated buffer). Before fragments can be sent, sending permission need to be requested through the link conges-

tion control (LCC) brick. If permission is granted, the sending process can continue. Otherwise, the fragment is kept in the buffer in order to be sent later.

After sending permission has been granted, the address of the next hop is retrieved through the network compartment or routing protocol. The next hop address is retrieved for every single fragment, even if some of them belong to the same connection. This is because in mobile networks routes can change frequently. Once the next hop address has been acquired, a communication channel to the next hop is requested through the network compartment and the fragment is sent. (Read section 2.3.2 for details on this communication channel.) Note that before sending, fragments may need to be split into smaller data units, so-called packets, in order to respect the network compartment's MTU (Maximum Transmission Unit). The conversion of fragment to packets and vice versa works identically to that of segment to fragments (see 3.3).

Note: The fragment buffering system is implemented in the hop-by-hop main brick, but this brick only stores the fragments. The brick actually managing this buffer system is the LCC brick. This means, retransmission, fragment deletion and buffer flushing are ordered directly by the LCC brick.

Receiving Data
Every fragment received is immediately acknowledged to its previous hop, i.e., the node that sent the fragment. (Duplicated fragments must also be acknowledged in case a previous acknowledgement was lost.) Afterwards the procedure depends upon the destination of the received fragment: if the fragment has arrived at its final destination, it is handed over to the end-to-end sublayer through the sublayer interface for delivery. If the fragment needs to be forwarded to another node it undergoes the store and forward procedure explained above.

Note: Fragment acknowledgements are also received by this brick. However, they are not processed locally but are passed on to the LCC brick as soon as they arrive.

3.4.2 Link Congestion Control Brick

This brick implements congestion control for the hop-by-hop sublayer. It runs on every intermediate node, i.e., every node on the route of the connection. The brick design is explained below.

Congestion Control
This brick limits the transfer on a link and thus, is active on the sending side of a link. It limits the transfer to the next hop by allowing only a certain number of unacknowledged fragments. It should be remembered that fragments may take different routes towards their final destination and thus do not arrive continuously at every node.

Every time the hop-by-hop main brick wants to send a fragment, sending permission has to be requested at this brick. If the link to which the fragment belongs to has not exceed the number of unacknowledged fragments, permission is granted. Otherwise, permission is denied and a transmission order for that fragment is sent to the hop-by-hop main brick as soon as a older fragment is acknowledged.

Link Management
To implement link congestion control efficiently, a system is necessary to keep track of the state of every fragment and the number of unacknowledged fragment on each link.

Fragment Acknowledgements
All fragment acknowledgements are received by the hop-by-hop main brick but are passed to this brick for processing. The acknowledgements are used to update the status of the corresponding fragments and the counters of unacknowledged fragments of each link.

Fragment Retransmissions
Specially on wireless mobile networks link failures occur often leading to packet loss. This is way fragments may need to be retransmitted. The link management system should implement timers for each fragment in order to check if a fragment has not been acknowledged after its timeout. Retransmission on the hop-by-hop sublayer should always occur in the same time intervals, i.e., not in exponentially increasing back-off intervals as for the end-to-end sublayer. This is because the cause of packet loss on the hop-by-hop sublayer is due to link failure and not due to congestion.

Additional Features
To increase performance and optimize resources the following functionalities can also be implemented:

- Estimation of sending rate to next hop with the use of measured transfer time of each fragment to limit transfer more accurately. (To be used in conjunction with the information provided by the link flow control

brick)

- Determine an adequate fragment buffer size according to usage

3.4.3 Link Flow Control Brick

This brick implements flow control for the hop-by-hop sublayer. However, the implementation should allow this brick to share its information on links with several protocols and eventually even gather feedback through them. The tasks of this brick are as follows:

Link Flow Control
This brick runs on the sending side of a link and counts outgoing packets and bytes to determine a link's rate. With the help of that information, the brick can also estimate if a hop has moved away or not. The information gathered by this brick is to be shared with all connections using the involved links upon request. In the context of SAFT for ANA, this brick will mainly be useful for the LCC brick.

3.5 Comparison to legacy network implementation

To ease the implementation of the above design on ANA, some differences to an implementation on a legacy network architecture should be recalled.

ANA allows dynamic protocol stacks, i.e., there is no strict layer system as known from the OSI reference model [OSIRM]. This means, SAFT can be implemented in a way that it also works without the functionalities of a network protocol when those functionalities are not available or are not needed. For example, SAFT could be run directly over Ethernet instead of IP, thus skipping the network layer of traditional architectures.

Significant is also the fact that ANA uses *targets* and *contexts* to specify which brick is to be found where. The effect this has on an implementation of SAFT is that the header of a packet will not have a fixed length as the source and destination of a packet will contain target and context fields which are strings of variable length. (For a detailed description of the terms target and context see section 4.2.1.)

4 Implementation

The current implementation of SAFT uses the design proposed in section 3 as a guideline. For the implementation of the header, [SAFT] is used as a reference with some adaptations for ANA. Several tools provided by the ANA API (see [ANAC]) are used as well. The programming language used for SAFT and most implementations running on the ANA prototype is *C*.

The following sections give an overview of the state of the current implementation and discuss the functionalities implemented in the bricks of the SAFT compartment.

4.1 Overview: Current State

At this point, SAFT provides semi-reliable transport for a single connection. Semi-reliable means that data is acknowledged for each link, but not for the whole connection. This means, an intermediate node receives acknowledgement messages from its next hop, but the source node never receives acknowledgements from the destination node.

Nonetheless, the current functionality is provided by four bricks (see fig. 7): the end-to-end main brick, the sublayer interface, the hop-by-hop main brick and the link congestion control brick. The latter implements the intelligence needed to provide semi-reliability whereas the other three process the application data. At this stage, the above mentioned bricks only implement partially the functionalities specified in their design.

To avoid confusion, several synonyms for the implemented bricks discussed in the following sections are listed below:

- *SAFT end-to-end main brick:* end-to-end main brick, end-to-end brick, saftEtE
- *SAFT sublayer interface brick:* sublayer interface, SLI brick, saftSLI
- *SAFT hop-by-hop main brick:* hop-by-hop main brick, hop-by-hop brick, saftHbH
- SAFT link congestion control brick: link congestion control brick, LCC brick, saftLCC

4.2 Compartment Specific Settings

In this section some implementational details concerning the whole compartment, i.e., all bricks, are explained. Most functions or settings relevant for

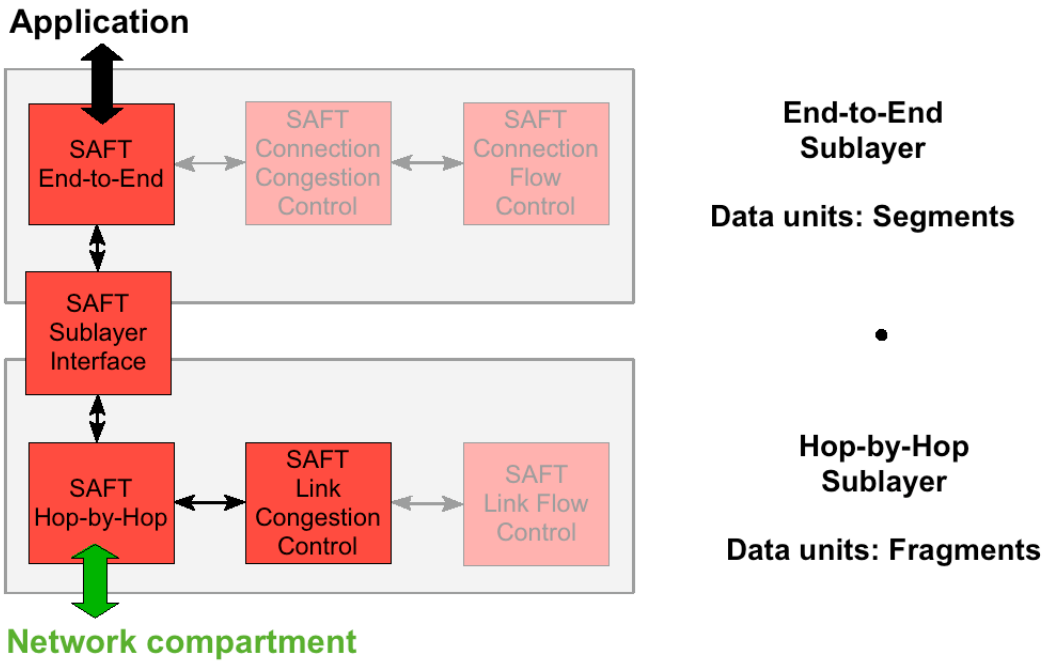

Figure 7: Currently implemented bricks (solid color) of SAFT compartment

the whole compartment can be found in the *saft.h* file. The most important are mentioned below.

4.2.1 SAFT Header

In order for the SAFT bricks to perform control mechanisms on the application data and to deliver it to the right nodes, all data units used by the SAFT compartment need to prepend a header to the application data they carry. When running SAFT on ANA the following header fields are necessary:

- source target and context
- destination target and context
- type
- segment number
- fragment number
- packet number

- segment acknowledgemente number
- fragment acknowledgemente number
- segment length (in number of fragments)
- fragment length (in number of packets)
- packet length (in bytes)

The header fields are explained below:

Source and Destination: In ANA the term *target* refers to a certain brick and the term *context* specifies where this brick is to be found. Pragmatically speaking, targets specify the applications sending or receiving data through the SAFT compartment and the context specifies the source and destination address (e.g., an IP address) of the nodes running those applications. This replaces the *address* and *port* fields, e.g., when running TCP or UDP over IPv4 on legacy network architectures.

To clarify this, source and destination for a http request are described: the source node running a web browser on IP address 10.0.0.2 sends a http request for a website running on the destination node's web server on IP address 10.0.0.1. In this example the web browser will ask the SAFT compartment to deliver its http request to the web server. The source and destination fields for the header of the SAFT segment carrying this http request are:

- source context: "10.0.0.2"
- source target: "webbrowser"
- destination context: "10.0.0.1"
- destination target: "webserver"

Type: The *type* field specifies if a data unit, i.e., a segment or a fragment, carries application data or control information (e.g., fragment or segment acknowledgements). This field is often refered to as *packet type*.

Segment, Fragment and Packet Numbers: These fields are used to order the application data. The segment number defines the order in which the application data was received at the source node allowing correct deliver at the destination node. The fragment number is used to retain the order of the data contained in the segments after they are split into fragments. The

packet number serves the same purpose when splitting fragments into packets before sending them through the network compartment.

Segment, Fragment and Packet Acknowledgment Numbers: These fields are only used in segment and fragment acknowledgements. They contain the number of the fragment or segment to be acknowledged to the sender of the original segment or fragment.

Segment, Fragment and Packet Length: The segment and fragment length fields are used when reassembling those data units. They are necessary for the receiving node to know how many fragments or packets to expect when reassembling segments and fragments respectively. The packet length field indicates the size in bytes of the payload that the packet carries.

In the current version of SAFT the header has been implemented the following way:

SAFT Header		
Field Name	**Explanation**	**Data Type**
srcContextLen	length of the source context string	uint8_t
srcContext	source context	char array
srcTargetLen	length of the source target string	uint8_t
srcTarget	source target	char array
destContextLen	length of the destination context string	uint8_t
destContext	destination context	char array
destTargetLen	length of the destination target string	uint8_t
destTarget	destination target	char array
type	type of data unit	uint8_t
segno	segment number	uint16_t
fragno	fragment number	uint8_t
pktno	packet number	uint8_t
segack	number of acknowledged segment	uint16_t
fragack	number of acknowledged fragment	uint8_t
seglen	length of segment in fragments	uint8_t
fraglen	length of fragment in packets	uint8_t
pktlen	length of packet in bytes	uint16_t

4.2.2 Segment and Fragment types

At this stage, the only packet types used by SAFT are: *data* and *fragment acknowledgement*. The type *segment acknowledgement* is defined but not yet used by any brick. The current packet *types* are define in saft.h with the following commands:

- #define SAFT_DATA 1
- #define SAFT_FRAGACK 2
- #define SAFT_SEGACK 3

4.2.3 XRP messages

To exchange information among bricks, XRP messages are used. XRP messages allow one to append meta data to the actual payload or information being sent. This is useful to specify the sender, the receiver, the type of content, etc., contained in a message. The most common XRP messages, mainly those implementing basic primitives, are included in the ANA API. However, SAFT implements some compartment specific XRP messages mainly to facilitate the communication between the link congestion control and the hop-by-hop main brick. The SAFT specific messages are explained in detail below. For more general information on XRP messages consult [ANAC, sec. 4.11]. Usage of basic primitives is explained in [ANAC, sec. 4].

- *Sending Permission Request:* This XRP message is used by the hop-by-hop main brick when requesting sending permission from the LCC brick for a specific fragment.

  Command Type: XRP_CMD_PERMREQUEST
  Number of Arguments: 1
  Argument Class: XRP_CLASS_SAFTHDR
  Argument: header of fragment to be sent, type struct saftHdr

- *Sending Permission Response:* This message is used by the LCC brick to answer a sending permission request received from the hop-by-hop main brick.

  Command Type: XRP_CMD_PERMRESPONSE

Number of Arguments: 1
Argument Class: XRP_CLASS_PERMISSION
Argument: permission , type uint8_t

permission can either be GRANTED or DENIED. Both are defined in saft.h as:

  - #define GRANTED 1
  - #define DENIED 2

- *(Re)Transmission Order:* This message is used by the LCC brick to order a transmission or a retransmission of a specific fragment from the hop-by-hop brick.

  Command Type: XRP_CMD_RESEND
  Number of Arguments: 1
  Argument Class: XRP_CLASS_ENTRYNAME
  Argument: name of fragment's entry in the fragment repository of saftHbH, type char[22]

- *Deletion Order:* With this message the LCC brick commands the hop-by-hop brick to delete a certain fragment in its local repository after a fragment has been acknowledged or resent a maximum number of times.

  Command Type: XRP_CMD_DELENTRY
  Number of Arguments: 1
  Argument Class: XRP_CLASS_ENTRYNAME
  Argument: name of fragment's entry in the fragment repository of saftHbH, type char[22]

Note: Because SAFT uses XRP messages bigger than most other bricks of the current ANA prototype, the XRP specs have to be modified. This is done with help of the *initSaftXRPSpecs()* function which is run at brick startup and is implemented in saft.h.

4.2.4 Application MTU

Currently in ANA there is no way to discover the MTU (Maximum Transmission Unit) of another brick automatically. If an application wants to send data messages using the SAFT compartment it can check the MTU_APP value, defined in the saft.h file, in order to find out the maximal message size. This value is important in order to know how big XRP messages will

be or to be able to set the size of fragments and packets to an optimal value. Definition:

- #define MTU_APP X
  where X is a size in bytes

4.2.5 Segment, Fragment and Packet Sizes

The size of fragments and packets needs to be adapted according to the network SAFT is used in. However, to determine the optimal values for different network scenarios is outside the scope of this thesis. These values are set in saft.h using the following commands, where X specifies a size in bytes:

- #define MAX_SAFTPKT_SIZE X
- #define MAX_SAFTFRAG_SIZE X

The size of a segment is determined by the application MTU and the header size. The header size is variable due to the fact that it contains context and target fields for the source and the destination which are strings of variable length. Nonetheless, considering 100 bytes as a maximum header size is sufficient in most cases. It allows for context and target strings to be up to 20 chars long if the space is distributed evenly. This is enough to use IPv4 and MAC addresses. Thus, the segment size does not exceed MTU_APP + 100 bytes in most cases.

4.3 End-to-End sublayer

This sublayer at the moment only implements one brick, the end-to-end main brick. Basically, it can manage one application request and perform the send and receive procedures necessary for the connection corresponding to the application request.

4.3.1 End-to-End main Brick

This brick's functions are implemented in the *saftEtE.c* file. Mainly these consist in handling a single application request, encapsulating the application data into segments and sending it. On the destination node the segments are reordered and delivered to the corresponding application. As this is the compartment provider brick, the above functionalities are shared using ANA primitives, thus the description of this brick's implementation focuses on the supported primitives, which currently are *publish*, *resolve* and *send*.

Essential functions
Here three functions are discussed which are essential for the brick to work.

brick_start(): This function must exist in every brick and is run at startup. It is used to initialize the brick. In the end-to-end main brick the initialization process comprehends the following tasks:

- initialize quick repository to buffer incoming segments (for QREP's see [ANAC, sec. 8])
- initialize an analock to control access to the above repository (for analocks see [ANAC, sec. 9])
- set the XRP Specs needed by SAFT using *initSaftXRPSpecs()*
- resolve the IP compartment to retrieve all local IP addresses and store them in a list
- publish the saftEtE brick in the sublayer interface in order to receive data from the hop-by-hop sublayer
- publish the saftEtE brick in the Minmex so other brick can find the SAFT compartment

entryPoint(): This function is bound to the IDP published in the Minmex. This means, publish and resolve request from applications will be received here. The task of this function consists in forwarding the requests to the corresponding functions handling them, i.e., *handlePublish()* or *handleResolve()*.

brick_exit(): This function must also be present in every brick. It is run on brick shutdown and is used mainly to free permanently allocated ressouces. In this brick the tasks performed at shutdown are:

- free permanently allocated IDP's
- free the quick repository initialized in brick_start()
- free the list containing the local IP addresses

Publish Primtive
After the saftEtE brick has published itself in the Minmex, other bricks can resolve the SAFT compartment and publish themselves in it. These will mostly be applications making themselves visible in the SAFT compartment in order to receive data through it.

Publish requests are handled by the *handlePublish()* function. Upon reception of a publish request, the IDP, through which the brick publishing itself wants to be reached, is stored in a permanently allocated IDP. No database or repository is yet implemented to handle multiple publish request as the remaining bricks of the SAFT compartment currently only support one application as well.

After the brick's IDP is stored, a reply to the publish request is sent back. If the publish request was handled successfully, all data destined to the published application received from the hop-by-hop sublayer through the sublayer interface is delivered. For further information on receiving data see the paragraph *Receiving Data* below.

See [ANAC, sec. 4.12.2] for general information on handling primitives.

Resolve Primtive
Through this primitive the actual functionality of the SAFT compartment is provided, i.e., a reliable communication channel to an application running on another node. At this stage, the communication channel only provides a semi-reliable connection, because control mechanisms are only implemented on the hop-by-hop sublayer. Nonetheless, this primitive allows an application to request a semi-reliable communication channel, i.e., an information channel using ANA terminology.

If an application wants to request an information channel through SAFT, the first step it has to take, is resolve the SAFT compartment to acquire the IDP bound to the entryPoint() function. Afterwards, it can send a *resolve* request to the SAFT compartment provider brick, i.e., this brick.

The resolve request received is handed over to the *handleResolve()* function where it is processed. Establishing an information channel requires the following steps:

1. extract the connection details out of the resolve request, i.e., source target and context and destination target and context of the brick to be resolved

2. request an IDP from the sublayer interface (using a resolve command) through which we will send the application data to the hop-by-hop sublayer

3. create and register an IDP to the callback function named *infochan()* which will handle the application data to be sent

   Important: when registering the callback function, all connection details are passed as arguments. This is needed for the callback function to have the necessary information for creating segment headers and send them to the right IDP requested from the sublayer interface in step 2.

4. send a reply to the brick that issued the resolve command containing the IDP bound to the callback function created in step 3 that will handle all application data to be sent

After this process, the application that sent the resolve command can send data to the target it resolved using the IDP it received in the resolve reply.

Note: The resolve request sent by the application triggers a "chain reaction" inside the SAFT compartment. As soon a resolve request is received, the end-to-end main brick resolves the sublayer interface and the sublayer interface resolves the hop-by-hop brick (see fig 8). This way, all functions on the data path can be accessed on data reception and do not have to be resolved first. Also, an IDP to the next brick only has to be resolved once, when the callback function is registered, and not every time data is received. Figure 9 shows the data path after the resolve process. Notice the new IDPs that were created in order to receive data.

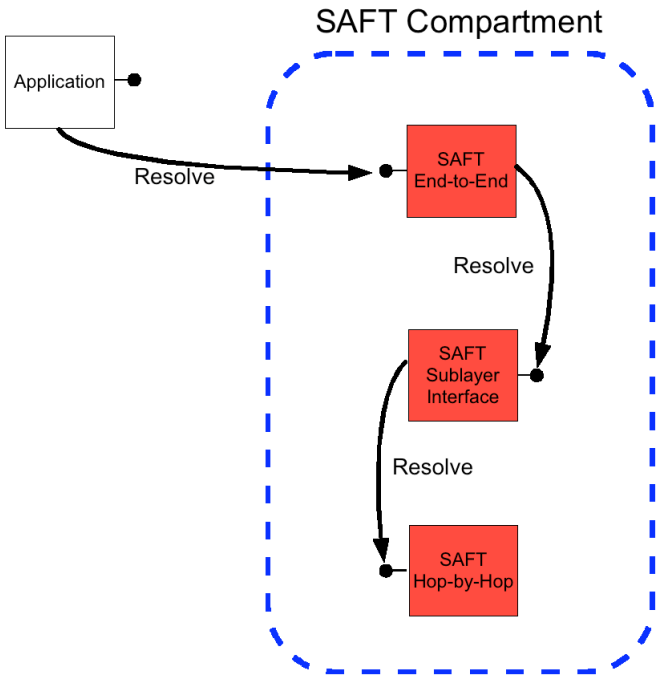

Figure 8: Chain reaction triggered by resolve request

Send Primtive

The *send* primitive is not received through the entryPoint() function like the publish and the resolve primitives. It is used by a brick that previously issued a resolve command to send data through the information channel created by SAFT.

As explained in the above paragraph, the resolve reply contains an IDP which can be used by the application requesting a communication channel to send data directly to the application it resolved. The *send* primitive is thus expected to be received at that IDP. The callback function to which that IDP is bound (*infochan()*) was previously registered when the resolve request was received.

In each brick a callback function named *infochan()* is implemented. It provides a part of the information channel's underlaying functionality and processes outgoing data. In this brick, the data received with every *send* command is processed in that callback function as follows:

1. the connection details which where passed to the callback function when registering it are retrieved and inserted into a segment header.

   Note: the application does not know its local context. The local con-

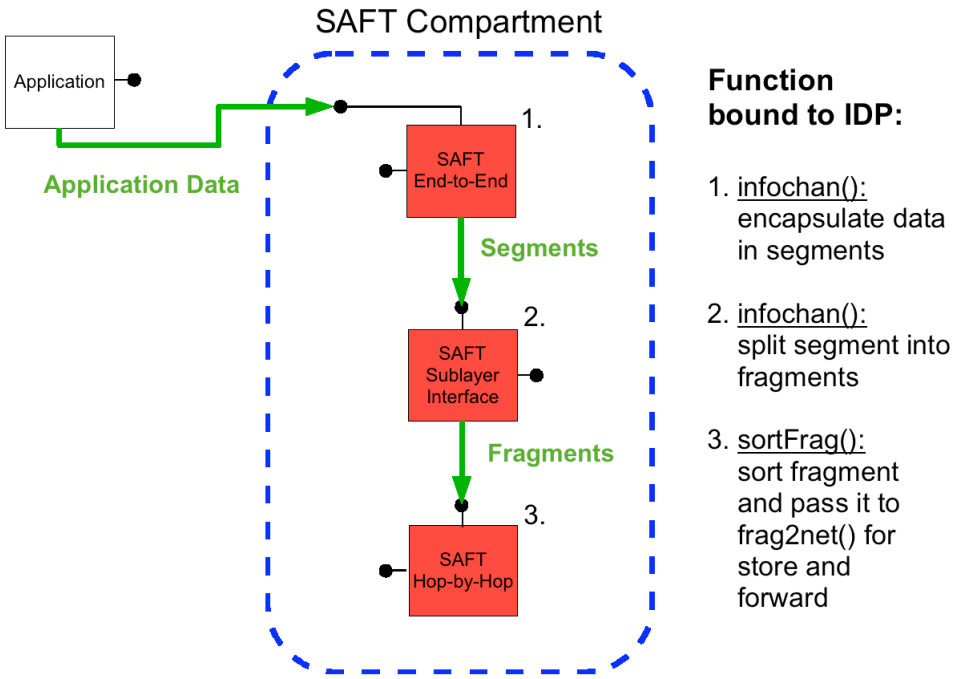

Figure 9: Data path after resolve process

text (e.g., local IP addresses) is retrieved through the IP compartment at startup (see brick_start() function) by this brick and stored in a list. The source context field is thus filled using the IP address stored in that list.

2. the missing header fields, i.e., segment number and packet type are added. As application data is processed here, packet type will always be SAFT_DATA.

   Note: header fields irrelevant for a data segment, e.g, fragment and packet number, are left blank. These are filled afterwards by other SAFT bricks when necessary.

3. the now complete segment header and the received application data are serialized, i.e., are converted into a unpadded sequence of bytes

4. the serialized segment is sent to the sublayer interface through the IDP which was passed to the callback function when it was registered

Now the sublayer interface will split the segment into fragments and hand them over to the hop-by-hop main brick which will send the data to the next

hop.

Receiving Data
In order to receive data from the hop-by-hop sublayer destined for applications, this brick publishes itself at startup in the sublayer interface (see brick_start() function). The IDP published in the sublayer interface is bound to the saftSLIRecv() callback function. All fragments received at the hop-by-hop sublayer reaching their destination node will be received at this callback function in form of segments after being process by the sublayer interface. The callback function handles incoming segments the following way:

1. deserialize received segment, i.e., reconstruct header and separate it from the application data

2. if the segment number is the next in line to be sent to the application, it is delivered along with all the succeeding continuous segments stored in the segment repository

3. else, stored segment in repository and send it as soon as its preceding segments have arrived

The above mechanism is implemented to order incoming segments, as it is possible that they arrive out of order due to packet loss on the connection routes.

4.4 Sublayer Interface Brick

The sublayer interface consists of one brick and is implemented in the *saftSLI.c* file. It fulfills the following tasks:

- split segments received from the end-to-end main brick into fragments and pass them to the hop-by-hop main brick

- receive fragments arriving at their final destination from the hop-by-hop main brick and reassemble them into segments for the end-to-end main brick

These tasks are achieved with three implemented primitives: *publish*, *resolve* and *send*. How the primitives and other functions of this brick are implemented is explained below.

Essential functions
Here three functions essential for the brick to work are explained:

brick_start(): This function must exist in every brick and is run at startup. It is used to initialize the brick. In the sublayer interface brick the initialization process comprehends the following tasks:

- initialize a quick repository to buffer incoming fragments
- initialize a lock to control access to the fragment repository
- set XRP specs needed by the SAFT compartment with the initSaftXRP-Specs() function
- publish the saftSLI brick in the hop-by-hop main brick in order to receive data from it destined to the end-to-end sublayer
- publish this brick in the Minmex so other bricks, e.g., the saftEtE brick, can resolve us

entryPoint(): This function is bound to the IDP published in the Minmex. This means, publish and resolve request will be received here. The task of this function consists in forwarding the requests to the corresponding functions handling them.

brick_exit(): This function must also be present in every brick. It is run on brick shutdown and is used mainly to free permanently allocated ressouces. In this brick the tasks performed at shutdown are:

- free permanently allocated IDP's
- free the quick repository initialized in brick_start()

Publish Primtive
This primitive is used mainly by the end-to-end main brick. The saftEtE brick publishes itself in the sublayer interface in order to receive data destined for the end-to-end sublayer coming from the hop-by-hop sublayer. The publish requests are handled by the *handlePublish()* function. The tasks performed within this function upon reception of a publish request are as follows:

1. extract the IDP on which saftEtE wants to be reached out of the XRP messaged received and store it in a permanently allocated IDP (no support for multiple publish requests is yet implemented)
2. send a reply confirming the publish request

After this process, all data received for the end-to-end sublayer can be sent to the IDP the saftEtE brick used to publish itself.

Resolve Primtive
The resolve primitive will also be used mainly by the saftEtE brick. The end-to-end main brick sends a resolve request to the sublayer interface every time it receives a resolve request from an application. The IDP contained in the resolve reply is used by the end-to-end main brick to send segments containing application data to the hop-by-hop sublayer. The resolve request received at this brick is processed by the *handleResolve()* function. The tasks performed by that function are detailed below:

1. request an IDP from the hop-by-hop main brick (using a resolve command) through which we will send the data received from the end-to-end sublayer to the hop-by-hop sublayer

2. create and register an IDP to the callback function named *infochan()* which will handle the segments received from the end-to-end main brick

   Important: when registering the callback function, the IDP we "resolved" in step 1 needs to passed as an argument. The infochan() function then can retrieve that IDP upon reception of segments form the end-to-end sublayer and send them converted as fragments to the hop-by-hop main brick.

3. send a resolve reply to the saftEtE brick containing the IDP bound to the callback function registered in step 2

After the above process, segments sent by the end-to-end to the sublayer interface are automatically passed to the hop-by-hop main brick as fragments after being process by the infochan() function of this brick.

Send Primtive
After the end-to-end main brick sent a resolve request to this brick, it can send segments to be processed by the sublayer interface using the *send* primitive. The segments are sent to the IDP which was received as a reply to the resolve request. The callback function bound to that IDP is the *infochan()* function. The tasks performed by that function upon receiving a segment are explained below:

1. deserialize the received segment

2. calculate how many fragments will be needed to carry the segments payload and the header. (It is to be remembered that the header is of variable length and that fragments have a fixed size. Thus the space available for payload must be calculated for every segment.)

3. create the necessary fragments by splitting the application data carried by the segment into several chunks and appending a header to them. The fragment number field of the header is filled in according to order of the application data.

4. send the created fragments to the hop-by-hop main brick through the IDP stored in the callback function's parameters. (That IDP was retrieved when this brick received a resolve request from the end-to-end main brick.)

Receiving Data
This brick receives all fragments that have reached their final destination from the hop-by-hop sublayer in order to reassemble them into segments and hand them over to the end-to-end sublayer. Before data can be received, this brick has to publish itself in the hop-by-hop main brick. This is done at startup in the brick_start() function. If the publish request in the hop-by-hop main brick was successful, all fragments are delivered to the IDP which is bound to the *saftHbHRecv()* function. There fragments are processed the following way:

1. store received fragment in fragment repository

2. check if all fragments belonging to the same segment have been received, i.e., the segment is complete

3. if segment is complete, reassemble it and send it to the end-to-end main brick using the IDP with which that brick published itself in the sublayer interface. (After sending, fragments in the repository belonging to the reassembled segment are deleted)

4. if segment is not yet complete, exit callback function

The above process is necessary to reorder the incoming fragments and convert them into segments before handing them over to the end-to-end main brick.

4.5 Hob-by-Hop sublayer

Currently this sublayer consists of two bricks implementing the store and forward procedure for a single connection. The hop-by-hop main brick implements link-based send and receive functionality, whereas the link congestion control brick implements the intelligence needed to provide a semi-reliable connection through the SAFT compartment.

4.5.1 Hop-by-Hop Main Brick

This brick is implemented in the *saftHbH.c* file. It interacts mainly with the sublayer interface, the link congestion control brick and the network compartment. The supported primitives include: *publish*, *resolve*, *send* and two compartment specific primitives used for congestion control. The implementation of this brick is detailed below.

Essential functions
Here three functions are discussed which are essential for the brick to work.

brick_start(): This function must exist in every brick and is run at startup. It is used to initialize the brick. In this brick, the initialization process comprehends the following tasks:

- initialize repository to buffer packets coming in through the network compartment
- initialize repository to store assembled fragments that need to be forwarded
- initialize analocks to control access to the above repositories
- set the XRP Specs needed by the SAFT compartment using *initSaftXRPSpecs()*
- publish this brick in the LCC brick in order to use congestion control
- publish this brick in the IP compartment to receive data through the network
- resolve the IP forwarding brick (ip_fwd) which is used to determine the next hop when forwarding fragments
- publish the saftHbH brick in the Minmex so other bricks can find it

- retrieve all local IP addresses through the IP compartment and store them in a list (needed to determine if a fragment is to be forwarded or delivered)

entryPoint(): This function is bound to the IDP published in the Minmex. This means, publish and resolve requests will be received here. The task of this function consists in forwarding the requests to the corresponding functions handling them. The primitives received from the the LCC brick are handled separately in the saftLCCRecv() function.

brick_exit(): This function must also be present in every brick. It is run on brick shutdown and is used mainly to free permanently allocated ressouces. In this brick the tasks performed at shutdown are:

- free permanently allocated IDP's
- free the repositories initialized in brick_start()
- free the list containing local IP addresses

Publish Primtive
In order for the sublayer interface to receive data destined for it, it has to publish itself in this brick. The data received by the sublayer interface are fragments that need to be reassembled into segments and handed over to the end-to-end main brick. *Publish* requests are handled by the *handlePublish()* function by performing the following tasks:

1. extract the IDP on which saftSLI wants to be reached out of the XRP messaged received and store it in a permanently allocated IDP (no support for multiple publish requests yet)
2. send a reply confirming the publish request

Resolve Primtive
This primitive is intended to be used mainly by the sublayer interface. For each resolve request the sublayer interface receives, it has to "resolve" an IDP from the hop-by-hop main brick in order to pass the data received from the end-to-end sublayer to the hop-by-hop sublayer. The *handleResolve()* function handles these request. Upon reception, a *resolve* request is handled the following way:

1. create and register an IDP to the callback function named *sortFrag()* which will handle the fragments received from the sublayer interface

Note: Unlike the other bricks discussed before, the hop-by-hop main brick does not resolve a communication channel upon reception of a resolve request. A communication channel is resolved separately for every fragment at transmission time because each fragment received may have a different next hop

2. send a reply to the saftSLI brick containing the IDP bound to the callback function created in step 1

After the resolve request was handled successfully, the saftSLI brick can send fragments destined for another node to the hop-by-hop main brick using the IDP created in step 1.

Send Primtive
The sublayer interface uses this primitive to send fragments to the hop-by-hop main brick. The fragments are sent to the IDP contained in a previous resolve request. That IDP is bound to the *sortFrag()* function of the hop-by-hop main brick and handles all incoming fragments. See paragraph *Handling Incoming Fragments* for further information.

Receiving Data through IP Compartment
In order for the SAFT compartment to receive data through the IP compartment, the hop-by-hop main brick publish itself in it at startup (see brick_start() function). The IDP that is published, is bound to the *ipRecv()* function. As mentioned in previous sections, fragments have to be split before sent through the network to respect the network compartment's MTU. Thus, data received through the network compartment is encapsulated in packets as well. The ipRecv() function implements a mechanism to reorder packets and reassemble them to fragments:

1. if received packet is of type fragment acknowledgement, send it to LCC brick and exit

2. else, store the received packet in the packet repository

3. check if all packets belonging to the same fragment have been received, i.e., the fragment is complete

4. if fragment is complete, reassemble it and send it to sortFrag() function for further processing

5. if fragment is not yet complete, exit the callback function

As stated in step 4, completed fragments are passed to the sortFrag() function which decides if a fragment has to be forwarded to another node or delivered locally. See *Handling Incoming Fragments* for more details.

Handling Incoming Fragments
All fragments received at this brick, either through the network compartment or from the end-to-end sublayer are received at the *sortfrag()* function. This function acknowledges all fragments necessary and afterwards sends those arriving at their final destination to the sublayer interface (using the IDP that brick published) and those destined for another node to the *frag2net()* function. The *frag2net()* function implements the *store and forward* procedure. Fragments received at this function undergo the following procedure:

1. store fragment in fragment repository
2. request sending permission for fragment through LCC brick
3. if permission is granted, retrieve next hop address using the IP forwarding brick and send fragment
4. exit function

In case sending permission is granted, an information channel is requested through the IP compartment by resolving the next hop. The resolve request contains the IP address of the next hop as destination *context* and *saftHbH* as destination *target*. Figure 10 shows an information channel between a source node and its next hop. One important detail to mention is that, the *channel type* argument of the resolve request must be set to 'm' (multicast) even if we only address one node. This is necessary because all other channel types establish a connection, which is not suitable for wireless mobile networks where frequent link failures difficult the connection establishment procedure. It is better to send data directly, without establishing a connection, as it is done with a multicast channel.

Fragments which did not receive sending permission are ordered to be sent by the LCC brick once older fragments have been acknowledged. Those fragments that were sent, but have not been acknowledged before their timeout, are ordered to be resent by the LCC brick. It should be remembered that the fragment repository is managed by the LCC brick even though it is implemented in the saftHbH brick.

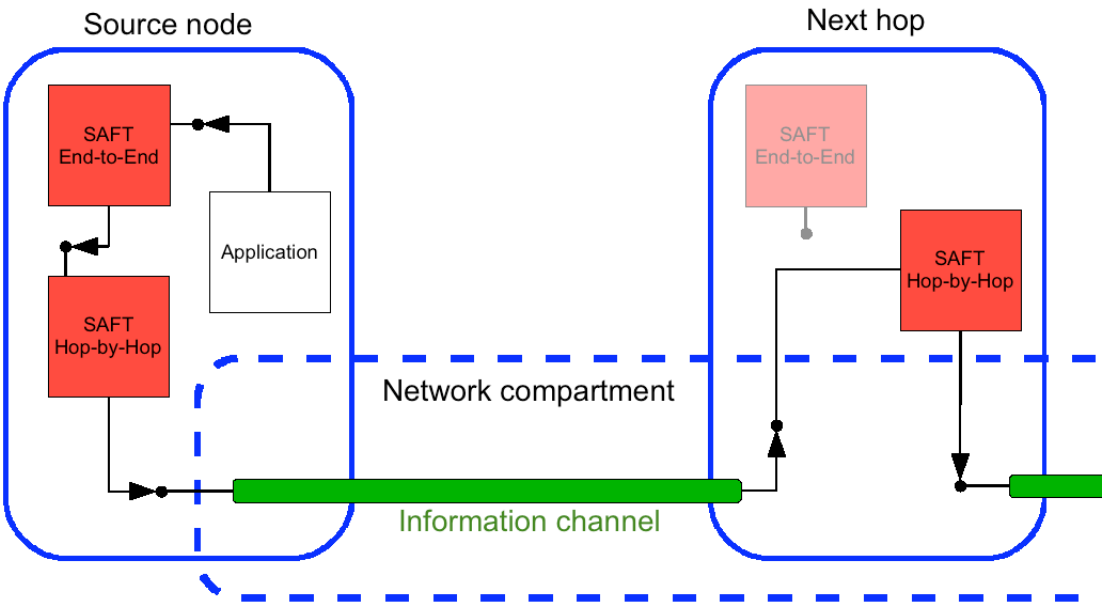

Figure 10: Information channel to next hop using IP compartment

Note: currently, fragments are acknowledged *selectively*, i.e., an acknowledgement is sent for every single fragment.

Handling Communication with Link Congestion Control
In order to handle the requests from the LCC brick, which manages the fragment repository, the saftHbH brick publishes itself in that brick on startup. After the publish request, all LCC messages are received at the *saftLCCRecv()* function. We expect two types of XRP messages: (re)transmission orders and deletion orders. Upon reception of such an order, the fragment specified in the message is retrieved from the fragment repository and (re)sent or deleted.

4.5.2 Link Congestion Control Brick

This brick provides congestion control for the hop-by-hop sublayer. It is implemented in the *saftLCC.c* and *saftLCC.h* files. It limits the transfer of the hop-by-hop main brick by granting or denying sending permission of fragments depending on the number of outstanding fragment acknowledgements. Details on the implementation are below.

Essential functions
Here three functions are discussed which are essential for the brick to work.

brick_start(): This function must exist in every brick and is run at startup.

It is used to initialize the brick. In this brick, the initialization process comprehends the following tasks:

- initialize repository to store the status of fragments
- initialize analocks to control access to the above repository and the unacknowledged fragments counter
- set the XRP Specs needed by the SAFT compartment using *initSaftXRPSpecs()*
- publish the saftLCC brick in the Minmex so other bricks can find it

  Note: this brick does not publish itself in any other brick

entryPoint(): This function is bound to the IDP published in the Minmex. The task of this function consists in forwarding the requests received to the corresponding functions handling them. Currently, *control messages*, *publish* requests and *sending permission* requests are expected.

brick_exit(): This function must also be present in every brick. It is run on brick shutdown and is used mainly to free permanently allocated ressouces. In this brick the tasks performed at shutdown are:

- free permanently allocated IDP's
- free the fragment status repositoriy initialized in brick_start()

Publish Primtive
The *publish* primitive will be used by the saftHbH brick. It publishes itself in this brick in order to be able to receive (re)transmission and deletion orders.

Handling Sending Permission Requests
As mentioned before, *sending pemission requests* are received through the IDP published in the minmex. They are processed in the *handlePermRequest()* function the following way:

1. extract the fragment header contained in the XRP message received
2. create a entry in the fragment status repository
3. check the unacknowledged fragments counter if sending permission can be granted or not

4. set a timer to check the status of the fragment after a specific timeout
5. create permission response XRP message
6. set the permission in the message to GRANTED if the number of unacknowledged fragments has not been exceeded, otherwise set it to DENIED
7. send the response message back to the saftHbH brick

The above procedure creates an entry in the fragment status repository for every fragment received. Currently the fragment status repository supports only one connection. Thus, the unacknowledged fragment counter is implemented as single variable.

Every time a new fragment is received, a timer is started that executes the *checkFragStatus()* function after a specific timeout. More details on updating a fragments status can be found in the paragrapgh *Updating Fragment Status*.

Handling Control Messages
All control messages of the hop-by-hop sublayer are received at this brick. The hop-by-hop main brick receives them through the network compartment and send them directly to this brick. They are processed in the *handleCtrlMsg()* function. Currently only fragment acknowledgements are expected. Upon reception of an acknowledgement the following tasks are performed:

1. mark corresponding fragment as acknowledged
2. decrease the unacknowledged fragment counter.

Updating Fragment Status
All fragments that have an entry in the fragment status repository have a corresponding timer activated. This timer checks the status of the corresponding fragment by executing the *checkFragStatus()* function after a specific timeout. Depending on the fragment's status, it is ordered to be (re)transmitted or deleted by sending a special XRP message (see sec. 4.2.3) to the hop-by-hop main brick. The *checkFragStatus()* function performs the following tasks upon receiving a fragment:

1. if fragment is unsent and counter of unacknowledged fragments has reached its maximum value:
   - reset timer
   - exit

2. if fragment is unsent but counter of unacknowledged fragments has *not* reached its maximum value:

   - change fragment status to unacknowledged
   - increase unacknowledged fragments counter
   - send a transmission order to saftHbH
   - reset timer
   - exit

3. if fragment is unacknowledged and it has *not* exceeded its maximum retransmission attempts:

   - increase its retransmission counter
   - send a retransmission order to saftHbH
   - reset timer
   - exit

4. if fragment is unacknowledged and it has exceeded its maximum retransmission attempts:

   - stop its timer
   - delete its fragment status repository entry
   - send a deletion order to saftHbH
   - decrease unacknowledged fragment counter
   - exit

LCC Settings
The LCC brick has several parameters that can be adjusted in the *saftLCC.h* file. These parameters can influence the performance and should be set according to each network. They are explained below:

- #define MAX_UNACKED_FRAGS X

  Defines the maximum number of unacknowledged fragments X allowed for a link.

- #define MAX_FRAG_RETRIES X

  Defines the maximum number of retransmission attempts X per fragment. If X is set to a value higher than 255, retransmission attempts will be unlimited.

- #define FRAG_CHECK_INTERVAL X

  Defines a time in miliseconds after which the timer of the fragment expires and the status of the fragment is checked. Keep in mind that the current ANA prototype has an internal timeout of 3 seconds that applies to all request done using basic primitives like publish and resolve.

4.6 Test Application: File Transfer

In order to test our SAFT implementation, an application was required. A simple tool to send a file from one node to another was therefor implemented. Its source code can be found in the *saftDemo.c* and *saftDemo.h* files. To send a file, the source node resolves the destination node through the SAFT compartment and sends the data to the IDP received in the resolve reply.

Once the SAFT compartment is running, to send a file the following steps have to be taken:

- on the destination node, start the saftDemo brick without additional command line parameters
- on the source node, start the saftDemo brick with the following parameters:

  saftDemo -n URL -c URL -d URL -a <destination> -a <file> -a <delay>

  The paramaters '-n', '-c' and '-d' specify the Minmex URL, the control gate URL and the data URL respectively. The additional parameters specify:

  - <destination>: IP address of the destination node, e.g., 192.168.0.10
  - <file>: path to the file to be sent, e.g., ./myfile.dat
  - <delay>: delay in seconds between to data messages sent. Useful to avoid flooding the SAFT compartment while no connection congestion control is implemented.

Additionally, the script in the *./scripts folder* can be used to start a node with all bricks necessary to run SAFT. The scripts also includes a simple shell script interface to send files using the saftDemo brick. This shell script also has a *testing mode* which performs the tests used for the validation.

The saftDemo brick creates a folder named *./saftDemoRecv* in the home directory of the user running it in order to store received files an write out

a log file (log.csv). The log file is a comma separated value table containing information on received files and all SAFT parameters set on brick start. This information can be useful for benchmarks and evaluations and was used for the validation of SAFT (see sec. 5).

5 Validation

A validation was performed to test the functionalities implemented in the SAFT compartment so far. The validation comprehends two test. Both tests and their results are presented below.

5.1 Overview

As a testing application the saftDemo brick was used (see sec. 4.6). Random files were created and sent through the network scenario depited below (fig. 11) simulating different levels of packet loss. Packet loss was simulated using *tc* (Traffic Control)for linux.

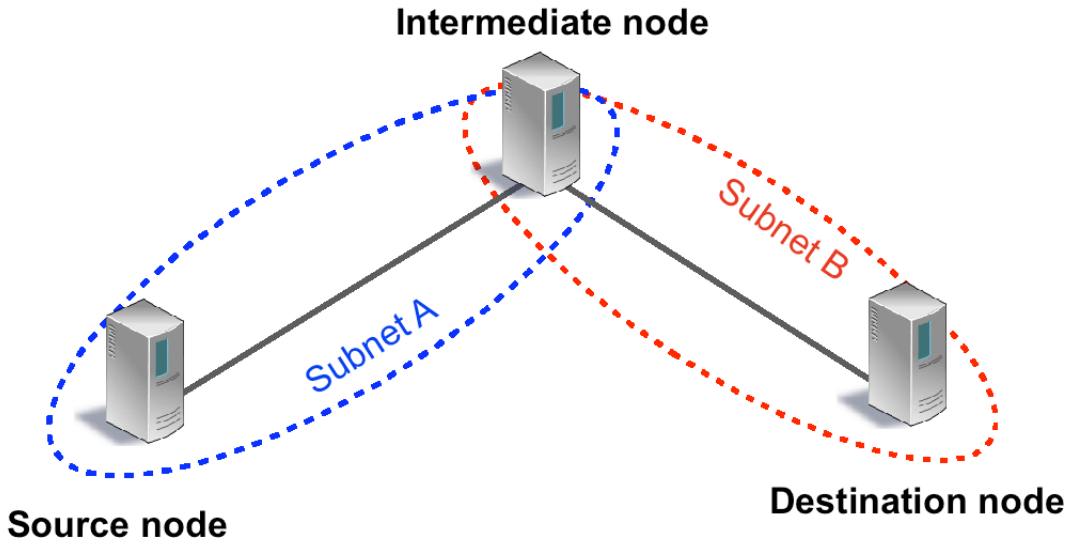

Figure 11: Network setup used for validation

The parameters used for the SAFT compartment during validation are listed below:

- application MTU: 10140B
- fragment size: 2560B
- packet size: 640B
- max. number of unacknowledged fragments: 10
- max. retransmissions of fragment: unlimited

- fragment retransmission interval: 3500ms

The saftDemo brick was setup to send a data message every 5 seconds. Considering the application MTU, this equals to an approximate transfer rate of 2 KB/s. This was necessary to avoid flooding the hop-by-hop sublayer as currently there is no connection congestion control implemented.

5.2 Test 1: Multiple Transmissions

For the first test a single random file, sized 30KB, was transmitted 100 times from the source node to the destination node over the network depicted in fig. 11. The test was done 4 times with different packet loss levels: no packet loss, 5% packet loss, 10% packet loss and 20% packet loss. Figure 12 shows the average transmission time for the different packet loss levels. Also, the relative increase compared to the non-lossy case is indicated on top of each column. An increase in average transmission time is visible: the higher the packet loss, the higher the average transmission time. However, the values of the increase have not been compared to mathematically expected values. Nonetheless, the test served to show that the SAFT compartment still delivered the file consistently after several successive transmissions.

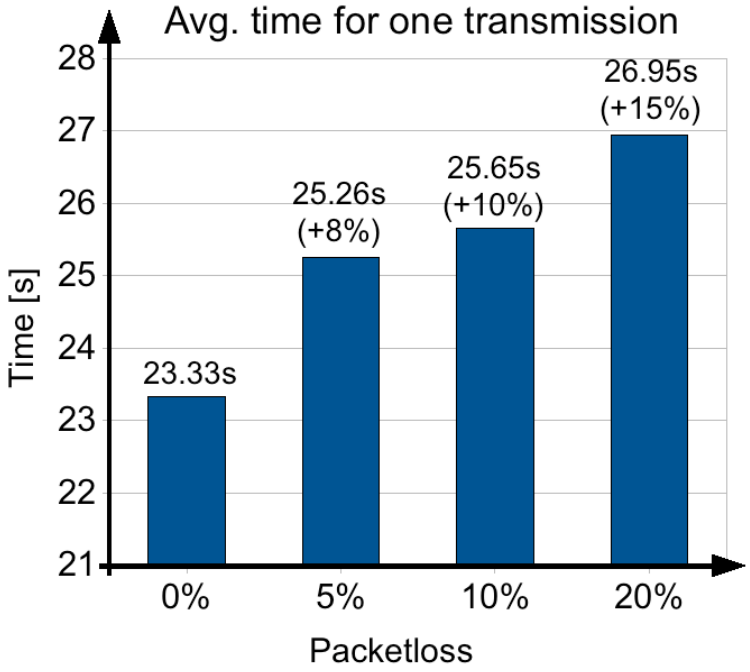

Figure 12: Avg. transmission time for 100 x 30KB

5.3 Test 2: Single Transmission

In this test, three random files, sizes 1B, 1.3KB and 14MB, were sent once from source to destination simulating the following packet loss levels: no packet loss, 5% packet loss and 10% packet loss. All files were received consistently for the different packet loss levels. Figure 13 shows the average transfer rate for the transmission of the 14MB file. The transfer rate for all packet loss levels is almost identical, due to the fact that the application's transfer rate creates s bottle neck. Thus, the influence of the packet loss is not visible. Despite the fact, this test showed that SAFT can handle a "larger" file and manage the involved congestion control properly.

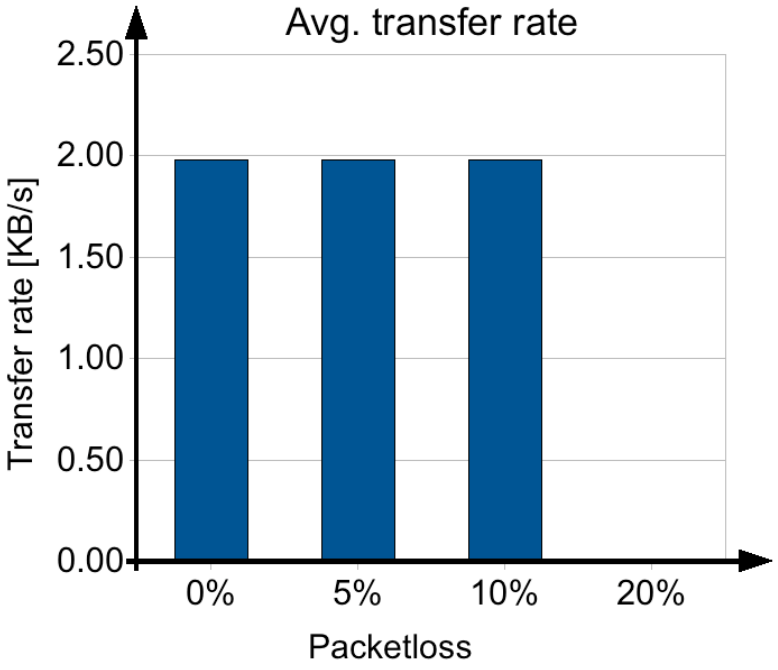

Figure 13: Avg. transfer rate for a 14MB file

6 Conclusion and Future Work

The paragraphs below summarize the work done in this thesis. Moreover, its relevance for the ANA project is considered, as well as the most important future work on SAFT for ANA.

6.1 Conclusion

With this semester thesis, the fundament for the first transport protocol to run on the ANA core architecture has been laid. As a hop-by-hop transport protocol, SAFT can provide great benefit for the ANA project through its strength on wireless mobile networks. It exemplifies that an architecture allowing easy incorporation of new protocols with non-traditional approaches can have great advantages.

Being the first transport protocol implemented, new possibilities are opened. A transport protocol providing reliability can be the motivation to create an interface for applications that run on legacy network architectures.

Also, by providing a flexible and modular design, the implementation of other transport protocols using SAFT as a groundwork has been eased. The design we proposed allows it to create basic versions of UDP and TCP with some modifications.

This initial implementation of SAFT provides semi-reliable communication channels. Thus, the next version of SAFT should focus on implementing connection congestion control to achieve full reliability and extend SAFT's range of applications. Nonetheless, the validation shows that, even though only a fraction of the SAFT design has been implemented, the current version of SAFT can provide a useful service for applications that work on semi-reliable communication channels.

6.2 Future Work

At current state, SAFT only implements a fraction of the bricks and functions proposed in our design. Hence, the next versions of SAFT should center on completing the implementation according to the design. However, the most urgent additions to be made are:

- **Connection Congestion Control**
  Currently SAFT only provides semi-reliable communication channels due to the fact that congestion control is only implemented on the hop-by-hop sublayer. This means, if data is lost on the route only the previous node and not the source node notices this. Additionally,

without congestion control on the end-to-end sublayer the application sending data through the SAFT compartment can "flood" the hop-by-hop sublayer as there is no mechanism stopping it from sending data or buffering the data.

- **Multiple Connection Support**
  At the moment, all bricks only support one connection. Repositories need to be implemented to handle multiple publish requests. The segment, fragment and packet buffers are implemented using quick repositories and packet numbers are used as repository entry identifiers. To store multiple connections in the same repository, a hash value could be computed from the header of each data unit and used as repository entry identifier.

Other, less urgent, topics to be addressed:

- **Variable Network Compartment**
  In the current implementation several functions only work in conjunction with the IP compartment. Ideally, the SAFT compartment should be able to work with *any* network compartment.

- **End-to-End Transportation**
  A modification of SAFT, implementing only end-to-end functionality, should be created in order to test the reusability of the bricks and the feasibility of an end-to-end protocol.

- **Performance Evaluation**
  A performance evaluation would certainly help to define SAFT's current capabilities and where optimization needs to be done.

References

[TLR] Simon Heimlicher, Rainer Baumann, Martin May and Bernhard Plattner. Computer Engineering and Networks Laboratory, ETH Zürich. January 2007

The Transport Layer Revisited

[SAFT] Simon Heimlicher. Master Thesis, April 2005, ETH Zürich.

SAFT: Store And Forward Transport. Reliable Transport in Wireless Mobile Ad-hoc Networks

[ANAB] Christian Tschudin (UBasel), Christophe Jelger (UBasel), Ghazi Bouabene (UBasel), Guy Leduc (ULg), Lorenzo Peluso (FOKUS), Manolis Sifalakis (ULancs), Marcus Schoeller (ULancs), Martin May (ETHZ), Matti Siekkinen (UOslo), Rudolf Roth (FOKUS), Stefan Schmid (NEC), Tanja Zseby (FOKUS), Thomas Plagemann (UOslo), Vera Goebel (UOslo)

ANA Blueprint - First Version Updated

[ANAC] Ghazi Bouabene (UBasel), Christophe Jelger (UBasel), Ariane Keller (ETHZ)

ANA Core Documentation D.1.8b

[ANAP] The ANA project's website. September 2008

http://www.ana-project.org

[IPCO] Stephan Dudler. Master Thesis, March 2008, ETH Zürich

New Protocols and Applications for the Future Internet

[RFC1122] R. Braden. Internet Engineering Task Force. October 1989.

RFC 1122

[OSIRM] John D. Day and Hubert Zimmermann. Proceedings of the IEEE, Vol. 71, No. 12. December 1983

The OSI Reference Model

[DPD] David D. Clark. Massachusetts Institute of Technology, Laboratory of Computer Science. 1988

The Design Philosophy of the DARPA Internet Protocols

[HBT] Zhenghua Fu, Xiaoqiao Meng, Songwu Lu. UCLA Computer Science Department. 2002

How Bad TCP Can Perform In Mobile Ad Hoc Networks

[CAC] Van Jacobson. University of California, Lawrence Berkeley Laboratory.

Congestion Avoidance and Control

A Appendix

A.1 Usage

To start the SAFT compartment the following bricks and compartments have to be running and configured:

- Minmex
- Vlink
- Ethernet compartment
- IP compartment
- RIP brick

After that, the SAFT bricks can be started. The bricks do not need any additional parameters. Only Minmex, control and data URL need to be specified. The bricks can thus be started the following way:

1. saftLCC -n <Minmex URL> -c <control URL> -d <data URL>
2. saftHbH -n <Minmex URL> -c <control URL> -d <data URL>
3. saftSLI -n <Minmex URL> -c <control URL> -d <data URL>
4. saftEtE -n <Minmex URL> -c <control URL> -d <data URL>

Ensure that the bricks are started in the order specified above, as they depend on each other.

Additionally, the scripts in the *./saft/scripts* folder can be used to setup a three node network using SAFT. The scripts in the *silentstart* subfolder include a shell script interface for using the saftDemo brick. See the *Readme files* in the ./scripts folder for instructions on running the different scripts.

A.2 Doxygen Documentation

A.2.1 SAFT End-to-End Main Brick

SAFT End-to-End Main Brick

Generated by Doxygen 1.5.5

Thu Sep 18 06:28:44 2008

Contents

1 File Index **1**

1.1 File List 1

2 File Documentation **3**

2.1 /Users/diego/Desktop/SA/saft/saftEtE.c File Reference 3

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

/Users/diego/Desktop/SA/saft/**saftEtE.c** 3

59

Chapter 2

File Documentation

2.1 /Users/diego/Desktop/SA/saft/saftEtE.c File Reference

```
#include "anaLib2.h"
#include "brick_template.h"
#include "./saft.h"
#include "quickRepository.h"
#include "analock.h"
#include "../ip/ip.h"
```

Defines

- #define **MODULE_NAME** saftEtE

Functions

- void AGENTCLASSMEMBER **entryPoint** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handlePublish** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handleResolve** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **saftSLIRecv** (struct anaL2_message ∗msg)
- NSSTATIC void AGENTCLASSMEMBER **infochan** (char ∗data, int len, anaLabel_t input, void ∗aux)
- struct fullTarget ∗ **cloneFullTarget** (struct fullTarget ∗ft)
- void **freeFullTarget** (struct fullTarget ∗ft)
- void AGENTCLASSMEMBER **brick_exit** ()
- int AGENTCLASSMEMBER **brick_start** ()

Variables

- char **mymodename** [ ] = "saftEtE"
- char ∗ **validatebuf**
- NSSTATIC anaLabel_t **entryPointIDP**

61

- NSSTATIC anaLabel_t **saftSLIIDP**
- NSSTATIC anaLabel_t **saftSLIRecvIDP**
- NSSTATIC anaLabel_t **appIDP**
- struct List **ipList**
- uint16_t **outgoingSegNum** = 1
- struct QREP **segRep**
- uint16_t **incomingSegNum** = 1
- analock_t **segRepLock**

2.1.1 Define Documentation

2.1.1.1 #define MODULE_NAME saftEtE

2.1.2 Function Documentation

2.1.2.1 void AGENTCLASSMEMBER brick_exit ()

2.1.2.2 int AGENTCLASSMEMBER brick_start ()

2.1.2.3 struct fullTarget* cloneFullTarget (struct fullTarget * *ft*) [read]

2.1.2.4 void AGENTCLASSMEMBER entryPoint (struct anaL2_message * *msg*)

This Callback Function handles the different requests from other bricks, e.g when a IDP is requested for data transmission trough a resolve command.

2.1.2.5 void freeFullTarget (struct fullTarget * *ft*)

2.1.2.6 void AGENTCLASSMEMBER handlePublish (struct anaL2_message * *msg*)

2.1.2.7 void AGENTCLASSMEMBER handleResolve (struct anaL2_message * *msg*)

2.1.2.8 NSSTATIC void AGENTCLASSMEMBER infochan (char * *data*, int *len*, anaLabel_t *input*, void * *aux*)

This function is bound to an IDP representing an information channel. The information channel IDP was created before through a resolve request of an application brick (e.g. an application wanting to send data using th SAFT compartment). This function is AL1 style.

2.1.2.9 void AGENTCLASSMEMBER saftSLIRecv (struct anaL2_message * *msg*)

This function is called when receiving segments from the sublayer interface that need to be sent to an application.

62

2.1.3 Variable Documentation

2.1.3.1 NSSTATIC anaLabel_t appIDP

2.1.3.2 NSSTATIC anaLabel_t entryPointIDP

2.1.3.3 uint16_t incomingSegNum = 1

2.1.3.4 struct List ipList

2.1.3.5 char mymodename[ ] = "saftEtE"

2.1.3.6 uint16_t outgoingSegNum = 1

2.1.3.7 NSSTATIC anaLabel_t saftSLIIDP

2.1.3.8 NSSTATIC anaLabel_t saftSLIRecvIDP

2.1.3.9 struct QREP segRep

2.1.3.10 analock_t segRepLock

2.1.3.11 char* validatebuf

A.2.2 SAFT Sublayer Interface

SAFT Sublayer Interface Brick

Generated by Doxygen 1.5.5

Thu Sep 18 06:26:18 2008

Contents

1 File Index **1**

1.1 File List 1

2 File Documentation **3**

2.1 /Users/diego/Desktop/SA/saft/saftSLI.c File Reference 3

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

/Users/diego/Desktop/SA/saft/**saftSLI.c** 3

69

Chapter 2

File Documentation

2.1 /Users/diego/Desktop/SA/saft/saftSLI.c File Reference

```
#include "anaLib2.h"
#include "brick_template.h"
#include "saft.h"
#include <math.h>
#include "quickRepository.h"
#include "analock.h"
#include <assert.h>
```

Defines

- #define **MODULE_NAME** saftSLI

Functions

- void AGENTCLASSMEMBER **entryPoint** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handlePublish** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handleResolve** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **saftHbHRecv** (struct anaL2_message ∗msg)
- NSSTATIC void AGENTCLASSMEMBER **infochan** (char ∗data, int len, anaLabel_t input, void ∗aux)
- void AGENTCLASSMEMBER **brick_exit** ()
- int AGENTCLASSMEMBER **brick_start** ()

Variables

- char **mymodename** [ ] = "saftSLI"
- char ∗ **validatebuf**
- NSSTATIC anaLabel_t **entryPointIDP** = NULL
- NSSTATIC anaLabel_t **saftHbHIDP** = NULL

71

- NSSTATIC anaLabel_t **saftHbHRecvIDP** = NULL
- NSSTATIC anaLabel_t **saftEtEIDP** = NULL
- struct QREP **fragRep**
- analock_t **fragRepLock**

2.1.1 Define Documentation

2.1.1.1 #define MODULE_NAME saftSLI

2.1.2 Function Documentation

2.1.2.1 void AGENTCLASSMEMBER brick_exit ()

2.1.2.2 int AGENTCLASSMEMBER brick_start ()

2.1.2.3 void AGENTCLASSMEMBER entryPoint (struct anaL2_message * *msg*)

This callback function handles the different requests from other bricks, e.g when a IDP is requested for data transmition trough a resolve command.

2.1.2.4 void AGENTCLASSMEMBER handlePublish (struct anaL2_message * *msg*)

2.1.2.5 void AGENTCLASSMEMBER handleResolve (struct anaL2_message * *msg*)

This function handels the resolve requests coming in through entryPointIDP. This will mainly be requests from the End-to-End sublayer brick wanting an information channel.

2.1.2.6 NSSTATIC void AGENTCLASSMEMBER infochan (char * *data*, int *len*, anaLabel_t *input*, void * *aux*)

This function is bound to an IDP representing an information channel. The information channel IDP was created before through a resolve request by a brick wanting to send data (e.g. the sublayer interface forwarding data from the End-to-End sublayer). This function is AL1 style.

Here we (Sublayer Interface) convert segments into fragments and pass them on to the SAFT Hop-by-Hop brick.

2.1.2.7 void AGENTCLASSMEMBER saftHbHRecv (struct anaL2_message * *msg*)

The following callback function handles the fragments coming in from the Hop-by-Hop sublayer/brick on the saftHbHRecvIDP. It converts them to segments and passes them on to the End-to-End brick (saftEtE). The order in which the segments are passed to the End-to-End is not defined.

72

2.1.3 Variable Documentation

2.1.3.1 NSSTATIC anaLabel_t entryPointIDP = NULL

2.1.3.2 struct QREP fragRep

2.1.3.3 analock_t fragRepLock

2.1.3.4 char mymodename[ ] = "saftSLI"

2.1.3.5 NSSTATIC anaLabel_t saftEtEIDP = NULL

2.1.3.6 NSSTATIC anaLabel_t saftHbHIDP = NULL

2.1.3.7 NSSTATIC anaLabel_t saftHbHRecvIDP = NULL

2.1.3.8 char* validatebuf

A.2.3 SAFT Hop-by-Hop Main Brick

SAFT Hop-by-Hop Main Brick

Generated by Doxygen 1.5.5

Thu Sep 18 06:19:20 2008

Contents

1 File Index **1**

1.1 File List 1

2 File Documentation **3**

2.1 /Users/diego/Desktop/SA/saft/saftHbH.c File Reference 3

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

/Users/diego/Desktop/SA/saft/**saftHbH.c** 3

79

Chapter 2

File Documentation

2.1 /Users/diego/Desktop/SA/saft/saftHbH.c File Reference

```
#include "anaLib2.h"
#include "brick_template.h"
#include "../ip/ip.h"
#include "saft.h"
#include <math.h>
#include "quickRepository.h"
```

Defines

- #define **MODULE_NAME** saftHbH

Functions

- void AGENTCLASSMEMBER **entryPoint** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handlePublish** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handleResolve** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **ipRecv** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **saftLCCRecv** (struct anaL2_message ∗msg)
- void **frag2net** (void ∗frag, unsigned int fragLen)
- int **sendFrag** (char ∗data, int len, anaLabel_t **ipEncIDP**, anaLabel_t **ipFwdIDP**)
- char ∗ **getNextHop** (anaLabel_t **ipFwdIDP**, char ∗ip)
- int **sendFragACK** (struct saftHdr ∗hdr, char ∗prevHop)
- NSSTATIC void AGENTCLASSMEMBER **sortFrag** (char ∗data, int len, anaLabel_t input, void ∗aux)
- void AGENTCLASSMEMBER **brick_exit** ()
- int AGENTCLASSMEMBER **brick_start** ()

81

Variables

- char **mymodename** [ ] = "saftHbH"
- char * **validatebuf**
- struct List **ipList**
- NSSTATIC anaLabel_t **entryPointIDP**
- NSSTATIC anaLabel_t **ipEncIDP**
- NSSTATIC anaLabel_t **ipRecvIDP**
- NSSTATIC anaLabel_t **ipFwdIDP**
- NSSTATIC anaLabel_t **saftLCCIDP**
- NSSTATIC anaLabel_t **saftLCCRecvIDP**
- NSSTATIC anaLabel_t **saftSLIIDP** = NULL
- struct QREP **pktRep**
- struct QREP **frags2netRep**
- analock_t **pktRepLock**
- analock_t **frags2netRepLock**

2.1.1 Define Documentation

2.1.1.1 #define MODULE_NAME saftHbH

2.1.2 Function Documentation

2.1.2.1 void AGENTCLASSMEMBER brick_exit ()

2.1.2.2 int AGENTCLASSMEMBER brick_start ()

2.1.2.3 void AGENTCLASSMEMBER entryPoint (struct anaL2_message * *msg*)

This callback function handles the different requests from other bricks, e.g when a IDP is requested for data transmition trough a resolve command.

2.1.2.4 void frag2net (void * *frag*, unsigned int *fragLen*)

This Function handles all fragments that need to be sent into the network and does the actual "store and forward". The fragments to be sent come from the from the **sortFrag()** (p. 6) function. A QREP is used to buffer the outgoing fragments. Sending permission for each fragment is requested from link congestion control (saftLCC). If LCC denies permission fragments is kept in the QREP for later use.

2.1.2.5 char * getNextHop (anaLabel_t *ipFwdIDP*, char * *ip*)

This function is used to find out the next hop of a route to certain IP address using the IP forwarding brick. Do not forget to free the pointer returned as memory allocation is needed for return value.

Parameters:

ipFwdIDP IDP through which we can reach the forwarding brick

ip The IP address for which we need the next hop

Returns:

IP address of next hop as char pointer or NULL on error

82

2.1.2.6 void AGENTCLASSMEMBER handlePublish (struct anaL2_message * *msg*)

2.1.2.7 void AGENTCLASSMEMBER handleResolve (struct anaL2_message * *msg*)

This function handles the resolve requests coming in through entryPointIDP. This will mainly be requests from the sublayer interface brick wanting to send us fragments.

2.1.2.8 void AGENTCLASSMEMBER ipRecv (struct anaL2_message * *msg*)

The following callback function handles the data coming in from the IP compartment on the ipRecvIDP.

Here we join the received packets into fragments and pass them to the **sortFrag()** (p. 6) function.

2.1.2.9 void AGENTCLASSMEMBER saftLCCRecv (struct anaL2_message * *msg*)

This function receives instruction from SAFT link congestion control. It executes requested retransmission of unacked fragments or deletes fragments in the local frags2net repository that were either successfully sent or timedout.

2.1.2.10 int sendFrag (char * *data*, int *len*, anaLabel_t *ipEncIDP*, anaLabel_t *ipFwdIDP*)

This function takes a fragment destined to another node, splits it into packets and sends it to the next hop. It also appends the local IP if no source context is specified.

Parameters:

data Fragment to be sent

len Size in bytes fo the fragment

ipEncIDP IDP of the ip_enc brick to which we send a resolve request for the next hop

ipFwdIDP IDP of the IP forwarding brick from which we retrieve the ip address of the next hop

Returns:

0 if everything went well, else -1

2.1.2.11 int sendFragACK (struct saftHdr * *hdr*, char * *prevHop*)

This function sends an ACK message for the fragment specified in the header to the previous hop that sent this fragment.

Important: For now we don't set the source Context of the ACK message. At current state, this does not have an influence, but it should be implemented in the future.

Parameters:

hdr SAFT header of the fragment to acknowledge

prevHop Context (e.g. IP address) of the node who sent the fragment

Returns:

1 on success, 0 else

2.1.2.12 NSSTATIC void AGENTCLASSMEMBER sortFrag (char * *data*, int *len*, anaLabel_t *input*, void * *aux*)

This function processes the received fragments and forwards them to another node or passes them on to the End-to-End layer for local delivery depending on ther final destination. The fragments can either come from the sublayer interface or from the **ipRecv()** (p. 5) callback function. This function also sends ACK's for non-local fragments.

Parameters:

aux We expect this to be the IP address of the previous hop in case we need to send a ACK message.

2.1.3 Variable Documentation

2.1.3.1 NSSTATIC anaLabel_t entryPointIDP

2.1.3.2 struct QREP frags2netRep

2.1.3.3 analock_t frags2netRepLock

2.1.3.4 NSSTATIC anaLabel_t ipEncIDP

2.1.3.5 NSSTATIC anaLabel_t ipFwdIDP

2.1.3.6 struct List ipList

2.1.3.7 NSSTATIC anaLabel_t ipRecvIDP

2.1.3.8 char mymodename[ ] = "saftHbH"

2.1.3.9 struct QREP pktRep

2.1.3.10 analock_t pktRepLock

2.1.3.11 NSSTATIC anaLabel_t saftLCCIDP

2.1.3.12 NSSTATIC anaLabel_t saftLCCRecvIDP

2.1.3.13 NSSTATIC anaLabel_t saftSLIIDP = NULL

2.1.3.14 char* validatebuf

A.2.4 SAFT Link Congestion Control Brick

SAFT Link Congestion Control Brick

Generated by Doxygen 1.5.5

Thu Sep 18 06:22:08 2008

Contents

1 Data Structure Index **1**

1.1 Data Structures 1

2 File Index **3**

2.1 File List 3

3 Data Structure Documentation **5**

3.1 fragStat Struct Reference 5

4 File Documentation **7**

4.1 /Users/diego/Desktop/SA/saft/saftLCC.c File Reference 7

4.2 /Users/diego/Desktop/SA/saft/saftLCC.h File Reference 10

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

fragStat 5

90

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

/Users/diego/Desktop/SA/saft/**saftLCC.c** 7
/Users/diego/Desktop/SA/saft/**saftLCC.h** 10

92

Chapter 3

Data Structure Documentation

3.1 fragStat Struct Reference

`#include <saftLCC.h>`

Data Fields

- uint8_t **retries**
- uint8_t **status**
- unsigned int **timerID**

3.1.1 Field Documentation

3.1.1.1 uint8_t fragStat::retries

3.1.1.2 uint8_t fragStat::status

3.1.1.3 unsigned int fragStat::timerID

The documentation for this struct was generated from the following file:

- /Users/diego/Desktop/SA/saft/**saftLCC.h**

94

Chapter 4

File Documentation

4.1 /Users/diego/Desktop/SA/saft/saftLCC.c File Reference

```
#include "anaLib2.h"
#include "brick_template.h"
#include "saft.h"
#include "quickRepository.h"
#include "anatimer.h"
#include <assert.h>
#include "saftLCC.h"
```

Defines

- #define **MODULE_NAME** saftLCC

Functions

- void AGENTCLASSMEMBER **entryPoint** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handlePublish** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handleCtrlMsg** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handlePermRequest** (struct anaL2_message ∗msg)
- void **checkFragStatus** (char ∗fragID)
- void AGENTCLASSMEMBER **brick_exit** ()
- int AGENTCLASSMEMBER **brick_start** ()

Variables

- char **mymodename** [ ] = "saftLCC"
- char ∗ **validatebuf**
- NSSTATIC anaLabel_t **saftHbHIDP**
- NSSTATIC anaLabel_t **entryPointIDP**
- struct QREP **fragStatRep**

96

- analock_t **fragStatRepLock**
- analock_t **UNACKED_FRAGS_lock**
- unsigned int **UNACKED_FRAGS** = 0

4.1.1 Define Documentation

4.1.1.1 #define MODULE_NAME saftLCC

4.1.2 Function Documentation

4.1.2.1 void AGENTCLASSMEMBER brick_exit ()

4.1.2.2 int AGENTCLASSMEMBER brick_start ()

4.1.2.3 void checkFragStatus (char * *fragID*)

This function is started by a timer in order to check the status of a fragment. If necessary it resends the fragment or orders the deletion of it in the saftHbH brick. It also updates the counter of unacked fragments if a fragment was sent successfully.

4.1.2.4 void AGENTCLASSMEMBER entryPoint (struct anaL2_message * *msg*)

This callback function handles the different requests coming mainly from the SAFT hop-by-Hop brick (saftHbH).

4.1.2.5 void AGENTCLASSMEMBER handleCtrlMsg (struct anaL2_message * *msg*)

This function handles the control messages of the Hop-by-Hop sublayer. The saftHbH bricks does not process them an passes them directly to this brick.

4.1.2.6 void AGENTCLASSMEMBER handlePermRequest (struct anaL2_message * *msg*)

This function handles the request for sending permission coming in from saftHbH. If we haven't exceeded the maximum number of unacknowledged fragments we grant permission to send fragment immediately. Otherwise, we set a timer to order retransmission of fragment afterwards.

97

4.1.2.7 void AGENTCLASSMEMBER handlePublish (struct anaL2_message * *msg*)

4.1.3 Variable Documentation

4.1.3.1 NSSTATIC anaLabel_t entryPointIDP

4.1.3.2 struct QREP fragStatRep

4.1.3.3 analock_t fragStatRepLock

4.1.3.4 char mymodename[ ] = "saftLCC"

4.1.3.5 NSSTATIC anaLabel_t saftHbHIDP

4.1.3.6 unsigned int UNACKED_FRAGS = 0

4.1.3.7 analock_t UNACKED_FRAGS_lock

4.1.3.8 char* validatebuf

98

4.2 /Users/diego/Desktop/SA/saft/saftLCC.h File Reference

Data Structures

- struct **fragStat**

Defines

- #define **STATUS_UNSENT** 0
- #define **STATUS_ACKED** 1
- #define **STATUS_UNACKED** 2
- #define **MAX_UNACKED_FRAGS** 10
- #define **MAX_FRAG_RETRIES** 10000
- #define **FRAG_CHECK_INTERVAL** 3500

4.2.1 Define Documentation

4.2.1.1 #define FRAG_CHECK_INTERVAL 3500

4.2.1.2 #define MAX_FRAG_RETRIES 10000

4.2.1.3 #define MAX_UNACKED_FRAGS 10

4.2.1.4 #define STATUS_ACKED 1

4.2.1.5 #define STATUS_UNACKED 2

4.2.1.6 #define STATUS_UNSENT 0

A.2.5 SAFT Main Include File

SAFT Main Include File

Generated by Doxygen 1.5.5

Thu Sep 18 06:32:44 2008

Contents

1 Data Structure Index **1**

1.1 Data Structures 1

2 File Index **3**

2.1 File List 3

3 Data Structure Documentation **5**

3.1 group Struct Reference 5

3.2 listelem Struct Reference 6

3.3 saftHdr Struct Reference 7

4 File Documentation **9**

4.1 /Users/diego/Desktop/SA/saft/saft.h File Reference 9

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

group 5
listelem 6
saftHdr 7

105

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

/Users/diego/Desktop/SA/saft/**saft.h** 9

107

Chapter 3

Data Structure Documentation

3.1 group Struct Reference

`#include <saft.h>`

Data Fields

- uint8_t * **safedUnits**
- void * **data**
- unsigned int **dataSize**

3.1.1 Detailed Description

Struct to hold a **group** (p. 5), i.e. their data and a unit counter

3.1.2 Field Documentation

3.1.2.1 uint8_t* group::safedUnits

which packets/fragments have already been received

3.1.2.2 void* group::data

3.1.2.3 unsigned int group::dataSize

this is needed because the actual size of a fragment/segment is not knwon until we receive its last packet/fragment

The documentation for this struct was generated from the following file:

- /Users/diego/Desktop/SA/saft/**saft.h**

109

3.2 listelem Struct Reference

`#include <saft.h>`

Data Fields

- char * **val**

3.2.1 Detailed Description

List element, mainly store all local IP's in our IP list

3.2.2 Field Documentation

3.2.2.1 char* listelem::val

The documentation for this struct was generated from the following file:

- /Users/diego/Desktop/SA/saft/**saft.h**

110

3.3 saftHdr Struct Reference

`#include <saft.h>`

Data Fields

- char * **srcContext**
- **saftHdrStrLen srcContextLen**
- char * **srcTarget**
- **saftHdrStrLen srcTargetLen**
- char * **destContext**
- **saftHdrStrLen destContextLen**
- char * **destTarget**
- **saftHdrStrLen destTargetLen**
- uint8_t **type**
- uint16_t **segno**
- uint8_t **fragno**
- uint8_t **pktno**
- uint16_t **segack**
- uint8_t **fragack**
- uint8_t **seglen**
- uint8_t **fraglen**
- uint16_t **pktlen**

3.3.1 Detailed Description

SAFT packet header

3.3.2 Field Documentation

3.3.2.1 char* saftHdr::srcContext

e.g. source IP or MAC address

3.3.2.2 saftHdrStrLen saftHdr::srcContextLen

3.3.2.3 char* saftHdr::srcTarget

e.g. the name of application sending the data

3.3.2.4 saftHdrStrLen saftHdr::srcTargetLen

3.3.2.5 char* saftHdr::destContext

e.g. destination IP or MAC address

111

3.3.2.6 saftHdrStrLen saftHdr::destContextLen

3.3.2.7 char* saftHdr::destTarget

e.g. destination application name

3.3.2.8 saftHdrStrLen saftHdr::destTargetLen

3.3.2.9 uint8_t saftHdr::type

type of data: DATA, ACK, NACK, etc.

3.3.2.10 uint16_t saftHdr::segno

contains segment ID

3.3.2.11 uint8_t saftHdr::fragno

contains fragment ID

3.3.2.12 uint8_t saftHdr::pktno

contains packet ID

3.3.2.13 uint16_t saftHdr::segack

last segment received by destination

3.3.2.14 uint8_t saftHdr::fragack

last fragment received by destination

3.3.2.15 uint8_t saftHdr::seglen

lenght of a segment in fragments

3.3.2.16 uint8_t saftHdr::fraglen

lenght of a fragment in packets

3.3.2.17 uint16_t saftHdr::pktlen

size of packet payload in bytes

The documentation for this struct was generated from the following file:

- /Users/diego/Desktop/SA/saft/**saft.h**

Chapter 4

File Documentation

4.1 /Users/diego/Desktop/SA/saft/saft.h File Reference

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "listAPI.h"
#include "ip_hdr.h"
```

Data Structures

- struct **saftHdr**
- struct **group**
- struct **listelem**

Defines

- #define **SRC_CONTEXT_LEN_SIZE** 1
- #define **SRC_TARGET_LEN_SIZE** 1
- #define **DEST_CONTEXT_LEN_SIZE** 1
- #define **DEST_TARGET_LEN_SIZE** 1
- #define **TYPE_SIZE** 1
- #define **SEGNO_SIZE** 2
- #define **FRAGNO_SIZE** 1
- #define **PKTNO_SIZE** 1
- #define **SEGACK_SIZE** 2
- #define **FRAGACK_SIZE** 1
- #define **SEGLEN_SIZE** 1
- #define **FRAGLEN_SIZE** 1
- #define **PKTLEN_SIZE** 2
- #define **SAFT_DATA** 1
- #define **SAFT_FRAGACK** 2
- #define **SAFT_SEGACK** 3

113

- #define **MAX_SAFTPKT_SIZE** 640
- #define **MAX_SAFTFRAG_SIZE** 2560
- #define **MTU_APP** 10140
- #define **XRP_CMD_PERMREQUEST** "prq"
- #define **XRP_CMD_PERMRESPONSE** "prp"
- #define **XRP_CMD_DELENTRY** "dle"
- #define **XRP_CMD_RESEND** "rsd"
- #define **XRP_CMD_SAFTCTRLMSG** "scm"
- #define **XRP_CLASS_PERMISSION** "prm"
- #define **XRP_CLASS_SAFTHDR** "shd"
- #define **XRP_CLASS_ENTRYNAME** "enm"
- #define **GRANTED** 1
- #define **DENIED** 2

Typedefs

- typedef uint8_t **saftHdrStrLen**

Functions

- int **saftHdrSize** (struct **saftHdr** ∗hdr)
- void ∗ **saftSerialize** (struct **saftHdr** ∗hdr, void ∗data, int dataLen, int ∗totSize)
- struct **saftHdr saftDeserialize** (void ∗pkt, int pktLen, void ∗∗data, int ∗dataLen)
- void **freeSaftHdr** (struct **saftHdr** ∗hdr)
- void **printSaftHdr** (struct **saftHdr** hdr)
- struct **saftHdr ft2saftHdr** (struct fullTarget ∗ft)
- void **initSaftXRPSpecs** ()
- void **freeGroup** (struct **group** ∗grp)
- void **initSafedUnits** (uint8_t ∗sU, unsigned int nbUnits)
- int **groupIsComplete** (uint8_t ∗sU, unsigned int nbUnits)
- void **freelist** (void ∗elem)
- int **ipIsLocal** (struct List ipList, char ∗ip)

114

4.1.1 Define Documentation

4.1.1.1 #define DENIED 2

4.1.1.2 #define DEST_CONTEXT_LEN_SIZE 1

4.1.1.3 #define DEST_TARGET_LEN_SIZE 1

4.1.1.4 #define FRAGACK_SIZE 1

4.1.1.5 #define FRAGLEN_SIZE 1

4.1.1.6 #define FRAGNO_SIZE 1

4.1.1.7 #define GRANTED 1

4.1.1.8 #define MAX_SAFTFRAG_SIZE 2560

4.1.1.9 #define MAX_SAFTPKT_SIZE 640

4.1.1.10 #define MTU_APP 10140

4.1.1.11 #define PKTLEN_SIZE 2

4.1.1.12 #define PKTNO_SIZE 1

4.1.1.13 #define SAFT_DATA 1

4.1.1.14 #define SAFT_FRAGACK 2

4.1.1.15 #define SAFT_SEGACK 3

4.1.1.16 #define SEGACK_SIZE 2

4.1.1.17 #define SEGLEN_SIZE 1

4.1.1.18 #define SEGNO_SIZE 2

4.1.1.19 #define SRC_CONTEXT_LEN_SIZE 1

4.1.1.20 #define SRC_TARGET_LEN_SIZE 1

4.1.1.21 #define TYPE_SIZE 1

4.1.1.22 #define XRP_CLASS_ENTRYNAME "enm"

4.1.1.23 #define XRP_CLASS_PERMISSION "prm"

4.1.1.24 #define XRP_CLASS_SAFTHDR "shd"

4.1.1.25 #define XRP_CMD_DELENTRY "dle"

4.1.1.26 #define XRP_CMD_PERMREQUEST "prq"

4.1.1.27 #define XRP_CMD_PERMRESPONSE "prp"

4.1.1.28 #define XRP_CMD_RESEND "rsd"

4.1.1.29 #define XRP_CMD_SAFTCTRLMSG "scm"

4.1.2 Typedef Documentation

115

Parameters:

grp Group to free

4.1.3.2 void freelist (void * *elem*)

Function to free our IP list and its elements

4.1.3.3 void freeSaftHdr (struct saftHdr * *hdr*)

Function used to free a SAFT header after calling **saftDeserialize()** (p. 13) or **ft2saftHdr()** (p. 12). This function only free the char pointer inside the header and not the header itself. If you allocated memory for the header struct you have to free it separately.

Parameters:

hdr SAFT header to free

4.1.3.4 struct saftHdr ft2saftHdr (struct fullTarget * *ft*) [read]

Function to extract the information of a fullTarget struct and create a SAFT header (struct **saftHdr** (p. 7)) with it. A fullTarget does no contain all the information needed to fill a SAFT header completely, so you will have to add the missing fields manually. Remember to free the return header after use with **freeSaftHdr()** (p. 12).

Parameters:

ft pointer to the fullTarget from which to extract the information

See also:

freeSaftHdr() (p. 12)

Returns:

Returns a SAFT header that contains the fullTarget's info

4.1.3.5 int groupIsComplete (uint8_t * *sU*, unsigned int *nbUnits*)

Function to check if a **group** (p. 5) is complete

Parameters:

sU safedUnits array to check

nbUnits The number of elements held in that array

116

4.1.3.6 void initSafedUnits (uint8_t * *sU*, unsigned int *nbUnits*)

Function to initialize a safedUnits array

Parameters:

sU The safedUnits array to initialize

nbUnits The number of elements held in that array

4.1.3.7 void initSaftXRPSpecs ()

This function changes the default XRP Specs. We need this to increase the number of bytes that we can send using an XRP message (e.g. with anaL2_forwardData()).

4.1.3.8 int ipIsLocal (struct List *ipList*, char * *ip*)

Function to determine if a certain IP addresses is local or not

Parameters:

list The list containing all local IP addresses

ip IP address to check

4.1.3.9 void printSaftHdr (struct saftHdr *hdr*)

Function to print out the contents of a SAFT header

Parameters:

hdr Input header to print out

4.1.3.10 struct saftHdr saftDeserialize (void * *pkt*, int *pktLen*, void ** *data*, int * *dataLen*) [read]

Given a segment, fragment or packet this function will fill a SAFT header with the information extracted from a packet and set pointers to the corresponding data. The data pointer will be set to point to the payload of the packet within the original data, i.e. the data does not get copied. Remember to free the header returned after use with **freeSaftHdr()** (p. 12).

Parameters:

pkt Pointer to the segment, fragment or packet to be deserialized

pktlen Lenght of the data unit (segment, fragment or packet)

hdr Pointer of a SAFT header to be filled with the extracted data

data Pointer to store the address of the extracted data (Hint: is actually a pointer to a pointer!)

dataLen Pointer to store the size of data

117

See also:

saftSerialize() (p. 14)
freeSaftHdr() (p. 12)

Returns:

SAFT header of packet, fragment or segment

4.1.3.11 int saftHdrSize (struct saftHdr * *hdr*)

Function that returns the size of the SAFT header in bytes.

4.1.3.12 void* saftSerialize (struct saftHdr * *hdr*, void * *data*, int *dataLen*, int * *totSize*)

Function that joins and serializes a SAFT header with its corresponding data and returns a pointer to the complete data unit (segment, fragment or packet including header). Remember to free the returned data after use.

Parameters:

hdr Pointer to header of segment, fragment or packet

data Pointer to the data that is to be appended to the header

dataLen Lenght in bytes of the data

totSize Address that will contain total size of segment,fragment or packet

Returns:

Pointer to a complete segment,fragment or packet containing a SAFT header

A.2.6 SAFT Demo Brick

SAFT Demo Brick

Generated by Doxygen 1.5.5

Thu Sep 18 06:36:22 2008

Contents

1 File Index **1**

1.1 File List 1

2 File Documentation **3**

2.1 /Users/diego/Desktop/SA/saft/saftDemo.c File Reference 3

2.2 /Users/diego/Desktop/SA/saft/saftDemo.h File Reference 6

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

/Users/diego/Desktop/SA/saft/**saftDemo.c** 3
/Users/diego/Desktop/SA/saft/**saftDemo.h** 6

124

Chapter 2

File Documentation

2.1 /Users/diego/Desktop/SA/saft/saftDemo.c File Reference

```
#include "anaLib2.h"
#include "brick_template.h"
#include <string.h>
#include "saftDemo.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <stdint.h>
#include <stdlib.h>
#include "saftLCC.h"
```

Defines

- #define **MODULE_NAME** saftDemo

Functions

- void AGENTCLASSMEMBER **entryPoint** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handlePublish** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **handleResolve** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **saftEtERecv** (struct anaL2_message ∗msg)
- void AGENTCLASSMEMBER **brick_exit** ()
- int AGENTCLASSMEMBER **brick_start** ()

Variables

- char **mymodename** [ ] = "saftDemo"
- char ∗ **validatebuf**
- NSSTATIC anaLabel_t **entryPointIDP**

126

- NSSTATIC anaLabel_t **saftEtEIDP**
- NSSTATIC anaLabel_t **saftEtERecvIDP**
- FILE ∗ **recvFile**
- char ∗ **recvDir**
- char ∗ **filename**
- FILE ∗ **logfile**
- char ∗ **logfileName**
- unsigned long int **totalBytes**
- unsigned long int **dataBytes**
- struct timeval **start_time**
- struct timeval **finish_time**
- uint8_t **sending_interval**
- unsigned long int **dataMsgCounter**

2.1.1 Define Documentation

2.1.1.1 #define MODULE_NAME saftDemo

2.1.2 Function Documentation

2.1.2.1 void AGENTCLASSMEMBER brick_exit ()

2.1.2.2 int AGENTCLASSMEMBER brick_start ()

2.1.2.3 void AGENTCLASSMEMBER entryPoint (struct anaL2_message ∗ *msg*)

This callback function handles the different requests from other bricks.

2.1.2.4 void AGENTCLASSMEMBER handlePublish (struct anaL2_message ∗ *msg*)

This function handles the publish requests coming in through entryPointIDP.

2.1.2.5 void AGENTCLASSMEMBER handleResolve (struct anaL2_message ∗ *msg*)

This function handles the resolve requests coming in through entryPointIDP.

2.1.2.6 void AGENTCLASSMEMBER saftEtERecv (struct anaL2_message ∗ *msg*)

The following callback function handles the data coming in from the SAFT compartment. It will receive files sent by another saftDemo brick.

127

2.1.3 Variable Documentation

2.1.3.1 unsigned long int dataBytes

2.1.3.2 unsigned long int dataMsgCounter

2.1.3.3 NSSTATIC anaLabel_t entryPointIDP

2.1.3.4 char* filename

2.1.3.5 struct timeval finish_time

2.1.3.6 FILE* logfile

2.1.3.7 char* logfileName

2.1.3.8 char mymodename[ ] = "saftDemo"

2.1.3.9 char* recvDir

2.1.3.10 FILE* recvFile

2.1.3.11 NSSTATIC anaLabel_t saftEtEIDP

2.1.3.12 NSSTATIC anaLabel_t saftEtERecvIDP

2.1.3.13 uint8_t sending_interval

2.1.3.14 struct timeval start_time

2.1.3.15 unsigned long int totalBytes

2.1.3.16 char* validatebuf

2.2 /Users/diego/Desktop/SA/saft/saftDemo.h File Reference [128]

```
#include <stdint.h>
#include <string.h>
#include "saft.h"
#include <sys/time.h>
```

Defines

- #define **FILE_START** 1
- #define **FILE_DATA** 2
- #define **FILE_END** 3
- #define **DEFAULT_DELAY** 5

Functions

- void * **filestart** (char *__filename__, int *msgSize)
- void **sendFile** (char *dContext, char *__filename__)

2.2.1 Define Documentation

2.2.1.1 #define DEFAULT_DELAY 5

2.2.1.2 #define FILE_DATA 2

2.2.1.3 #define FILE_END 3

2.2.1.4 #define FILE_START 1

2.2.2 Function Documentation

2.2.2.1 void* filestart (char * *filename*, int * *msgSize*)

This function is used to create a file start (FILE_START) message. This is the first message we send when starting a file transfer.

Parameters:

filename full path name of file that is going to be sent

msgSize pointer that will be filled witht the size of the FILE_START message

2.2.2.2 void sendFile (char * *dContext*, char * *filename*)

This function will send a file to another instance of saftDemo running on a specified context/IP-address.

A.3 Assignment

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

130

Semester Thesis

Implementation of SAFT on ANA

Diego Adolf

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisors: Simon Heimlicher, heimlicher@tik.ee.ethz.ch
Dr. Martin May, may@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

March 2008 - June 2008

1 Introduction

This semester thesis is in the context of the ANA project. The goal of the ANA project is to explore novel ways of organizing and using networks beyond legacy Internet technology. The ultimate goal is to design and develop a novel network architecture that can demonstrate the feasibility and properties of autonomic networking. Many new protocols have been developed in the context of ANA.

In this thesis we focus on SAFT, a transport protocol designed for wireless networks. SAFT splits data into chunks and forwards these through the network independently. At every intermediate node, chunks are stored and then forwarded to the next node; thus SAFT does not depend on continuous end-to-end connectivity.

The objective of this semester thesis is to implement a basic version of SAFT within the ANA framework.

2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

2.1 Objectives

The goal of this semester thesis is to implement the SAFT transport protocol an ANA. The first step will be to design the software architecture. The architecture has to clearly divide the SAFT protocol in its elementary blocks and should allow its core mechanisms to be adapted or replaced in the future. In a second step SAFT will be implemented in ANA. The final step will be to evaluate the architecture and implementation.

2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

2.2.1 Familiarization

- Study the available literature on ANA [1, 2].

- Study the available literature on SAFT [3, 4].
- Familiarize yourself with the ana svn, the ana wiki and the ana developer mailing list.
- Setup a Linux machine on which you want to do your implementation, consider to use a vmware.
- Setup a test network (either physical or virtual) that runs the ANA chat application over the IP protocol (at least 3 nodes and 2 subnets).
- In collaboration with the advisor, derive a project plan for your semester thesis. Allow time for the design, implementation, evaluation, and documentation of your software.

2.2.2 Software Design

- Determine the areas where your implementation will touch other ANA parts, e.g. the forwarding table.
- Divide the SAFT protocol in suitable "Bricks". Each Brick should be able to perform an independent task.
- Consider how the protocol will be configured and fine tuned.
- Design a simple application that can test the implementation of SAFT.
- Think about possible test scenarios.

2.2.3 Implementation

- Determine the Bricks of your design which are needed for a minimalistic implementation of SAFT.
- Implement the minimal set of Bricks.
- Adhere to the Linux coding style guide [5].
- Provide a simple validation script, that determines whether your Bricks work correctly.
- Optional: If the minimal set of Bricks have been validated, implement the remaining Bricks.
- Optional: Adapt your implementation to runs in Linux kernel space and ns2 as well.

2.2.4 Validation

- Validate the correct operation of your implementation.
- Check the resilience of the implementation, including its configuration interface, to uneducated users.
- Optional: Do a performance evaluation of your implementation. What is the impact of the parameters? How well is link failure handled? etc.

2.2.5 Documentation

- Maintain an online documentation about the current status of your Bricks on the ana wiki [6].
- Document your code with doxygene according to the ANA guidelines [7].
- Write a documentation about the design, implementation and validation of SAFT in ANA.

3 Deliverables

- Provide a "project plan" which identifies the mile stones.
- Mid semester: Intermediate presentation. Give a presentation of 10 minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- End of semester: Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well. The final report must contain a summary, the assignment and the time schedule. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.
- The core source code will be published under the ISC license. Really smart Bricks will stay closed source.

5 References

[1] ANA Core Documentation: All you need to know to use and develop ANA software. Available in the ANA svn repository.
[2] ANA Blueprint: First Version Updated. Available from the ANA wiki
[3] Simon Heimlicher: SAFT - Store And Forward Transport: Reliable Transport in Wireless Mobile Ad-hoc Networks [TIK MA-2005-08]
[4] Severin Hafner and Rafael Schoenenberger: Implementation of SAFT [TIK SA-2006-08]
[5] Available on your Linux box: file:///usr/src/linux/Documentation/CodingStyle
[6] https://www.ana-project.org/wiki
[7] http://www.stack.nl/ dimitri/doxygen/