ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Stefan Lienhard

Design and Implementation of SAFT on ANA II

Semester Thesis SA-2008-21
September 2008 until January 2008

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

Abstract

The ANA prototype lacks a reliable transport protocol. In a previous semester thesis SAFT, a transport protocol designed to perform well on wireless networks, was partially implemented. We extend the hitherto incomplete SAFT implementation into a working version that meets all the basic requirements for a reliable transport protocol.

With the modular design the implementation serves as a basis for other transport protocols. With only small effort the existing modules could be used to create UDP or TCP for ANA.

Acknowledgements

I would like to thank my supervisor Prof. Dr. Bernhard Plattner for the opportunity of letting me write a thesis in the Communication Systems Group.

Furthermore I want to express my thanks to my two advisors Ariane Keller and Simon Heimlicher whose door was always open and who always provided assistance and support and guided me with inputs and feedback. Especially helpful were the Ariane's deep insights into the ANA framework. Simon as the inventor of SAFT helped me to better understand not only SAFT but transport protocols in general. I also express my thanks to Diego Adolf who wrote the forerunner thesis that was the foundation for my work. He was so kind to also let me use some of his graphics in my thesis.

Special thanks to go my friends for their emotional support that carried me through the endless and sleepless nights in front of the computer in ETZ G71.1. Particularly honored be Laura Bassi for constantly distracting me through the term.

Contents

Chapter 1

Introduction

The Internet as well as most of nowadays corporate and home networks run the *Internet Protocol Suite*. This suite is better known under the name of its prominent protocols, TCP/IP [1], [2]. The structure of this suite is a stack where every protocol only talks to two other protcols, the one above and the one beneath it. Each layer of the protocol stack allows the usage of multiple protocols but the layers cannot be reordered, removed or merged together in an easy way. Recent research and development have shown that this static structure that has been the standard for decades is not ideal anymore in many cases and that it is difficult to integrate new network types like mobile, ad-hoc or sensor networks.

The *Autonomic Network Architecture* (ANA) [3] aims to be a solution to those problems by providing a generic framework for testing and exploring novel ways of networking through flexible and dynamic formation of network nodes and whole networks. Modular composition makes it possible to build network architectures that are scalable. ANA's longterm goal would be to replace the current fixed network.

TCP has become the most used transport protocol but since it was originally designed for wired networks it does perform bad on wireless ones [4]. The control mechanisms in TCP assume that packet loss is mainly due to congestion which is not the case for wireless networks where most of the packet loss is blamed on link failures.

The SAFT protocol was designed to match wireless and mobile networks. It overcomes TCP's weaknesses and outperforms on the target networks (see [5], [6] and [7]). SAFT is a hop-by-hop transport protocol that stores the data on every intermediate hop. TCP on the other hand is a pure end-to-end approach.

In a recent semester thesis (*Implementation of SAFT on ANA* [8]) SAFT was partially implemented on the ANA framework by Diego Adolf. However, this implementation does not provide the full functionality of the SAFT protocol. The main missing features are reliability, congestion control and support for multiple applications. Hence in this thesis we extend the implementation with those features. With *SAFT v0.1* we refer to this old implementation whereas we call our enhanced version *SAFT v0.2*.

1.1 Main Goals

During the thesis the basic functionalities for a reliable transport protocol were to be implemented by extending the already existing SAFT v0.1 with:

- *End-to-end reliability*
  For a reliable transport protocol it is important that the sending side in a connection gets a confirmation when its data arrives at the destination. This is done by adding end-to-end reliability.

- *Congestion control*
  Some simple methods to avoid and handle congestion shall be implemented.

- *Support for multiple applications*
  Several applications should be able to use the protocol concurrently. In SAFT v0.1 only one application could use the protocol at a time but the user might want to check his email while surfing the web.

Chapter 2

Background Information and Related Work

This chapter introduces the backgrounds an the related work. We focus only on the essentials that are required to fully understand this thesis: the ANA core architecture, the design of the SAFT protocol, the ANA IP module (since we use IP as the underlying network protocol for the SAFT implementation) and the preceding thesis [8] that describes an initial implementation of the SAFT protocol for ANA.

2.1 Autonomic Network Architecture (ANA)

For an extended documentation we refer to the ANA core documentation [9] which is mainly a programmers guide and the ANA Blueprint [10] that describes the fundamental design and functioning in deep detail.

2.1.1 ANA Node

A ANA Node or an instance of ANA is divided into the *Minmex* an the *ANA Playground* (see Figure 2.1). The Playground is the part where all networking functionality is hold in modules whereas the Minmex is the control unit responsible for the communication between the modules. These modules are called *bricks* and are explained further in the next Section.

2.1.2 Bricks

The modules or bricks in the ANA Playground can be thought of as Lego pieces that need to be connected in the correct fashion to build a complete, self-contained network protocol or application. A complex protocol can be split up into smaller pieces by using the divide and conquer principle. This makes the code easier to understand since a brick is only handling one specific issue and the big advantage is code reusability. One brick can be used by several applications, for example a checksum brick (which is actually an existing part of the ANA IP layer, more information can be found in [2]) might be useful for miscellaneous purposes.

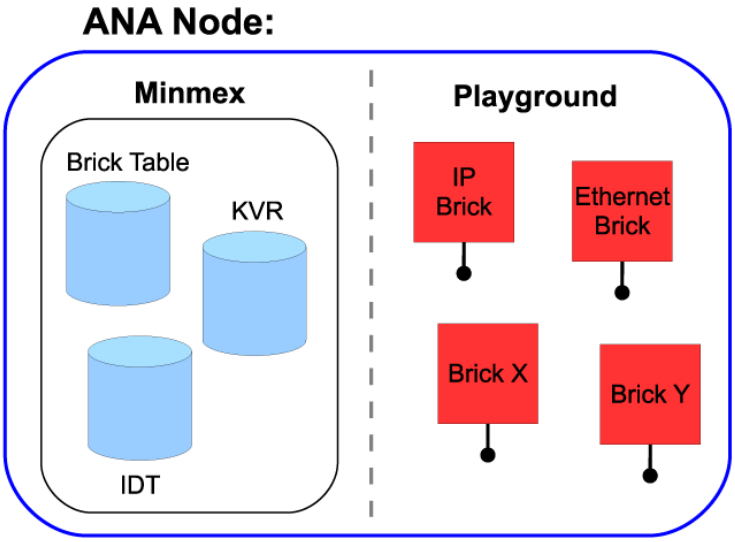


Figure 2.1: ANA node.

2.1.3 Compartment

According to [10, pp 8] 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." For our context this definition is too abstract. We can simplify it by defining a compartment as the group of all bricks on all nodes that are part of a certain application or protocol (often used compartments are e.g. the Ethernet or the IP compartment). For a more detailed and accurate description of the compartment concept we refer to Section 3.1 in [10].

Relevant for this thesis are only the IP compartment and of course the SAFT compartment which are the groups of all IP and all SAFT related bricks. Compartments can be overlaying or nested (represent a subset of another compartment).

Every node which is part of a certain compartment will also have a so-called compartment provider brick. It represents the compartment on the current node and is the gateway for bricks that are not part of the compartment to communicate with brick inside the compartment or to use the services offered by the compartment.

2.1.4 Information Channels and Information Dispatch Points

An *information channel* (IC) is an abstraction for a communication channel that is established between bricks or compartments.

An *information dispatch point* (IDP) is a gateway through which bricks can access other bricks. Bricks talk to each other and exchange data over IDPs. A brick can have an arbitrary number of IDPs and can create them during runtime. In graphics an IDP is always represented by a black dot connected with a line to the brick or the IC it is part of.

An example scenario that uses both ICs and IDPs: the IP compartment is

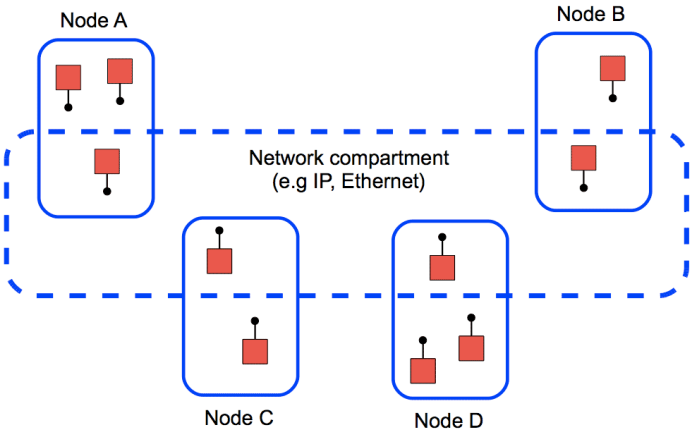


Figure 2.2: Nodes with bricks that are part of a network compartment. *(Image courtesy of Diego Adolf.)*

asked to make a connection to a network node with a certain IP address. As a result the IP compartment will establish an IC between from this local node to the machine with the mentioned address and will return an IDP which we use to communicate over the IC.

Implementation wise accessing an IDP is comparable with calling a callback function associated with that IDP. Those functions are specified by function pointers. If somebody wants to use the service provided by an IDP by sending data to it the callback function is called with the data as argument.

2.1.5 Primitives

Interaction between bricks over the IDPs is done using *primitives* [10, Section 3.3.3] which are fundamental paradigms of communication. Every compartment (respectively its compartment provider brick) needs to support some basic primitives for providing a generic way of communication between bricks, the most important are:

- *publish*
  This provides a certain service to other bricks. A service can be seen as a functionality offered by a certain brick. By publishing the IDP of a service to a compartment the service makes itself available to other bricks (for finding services see the next point). The publish primitive is needed for example when our brick expects data from an underlying compartment. By publishing ourselves to the compartment we made sure that bricks from that compartment can find us and know where to send their data to.

- *resolve*
  This primitive requests an IDP for a certain service or an IC to another node. Let's carry the example from the last point on. The publish of the receiving brick is not sufficient. Only the compartment knows about the receiving brick but the contained bricks do not know the receiver yet. So

before they can send data they have to ask their compartment (with a resolve primitve) for an IDP of the receiving brick.

- *send*
  Sends data to a certain IDP which was resolved before.enqueue

In terms of code a primitive is simply a function.

2.1.6 XRP Messages

The ANA uses XRP (*eXtensible Resolution Protocol* [11]) for internal control messages. XRP specifies a generic message format. Each message ha a *command* field which desribes the type or purpose of the message. Furthermore every message can have an arbitrary number of arguments of arbitrary type. This type is called *class*. Every arguments appended to a XRP message must also define the size of memory it requires.

ANA allows users to define own XRP commands and classes.

2.2 Store-and-forward Transport (SAFT)

For more detailed documentation we refer to [12], [8] and [6].

2.2.1 Overview

SAFT is a transport protocol that suites wireless and mobile networks by outperforming TCP on those network types. As mentioned in the introduction the control mechanisms in TCP assume that packet loss is mainly due to congestion which is not the case for target networks where most of the packet loss is caused by link failures. SAFT is a hop-by-hop protocol which buffers the data on every intermediate node.

SAFT is a transport protocol and would be running on the transport layer according to the OSI reference model [13]. SAFT splits the transport layer up into two loosely depending sublayers: the end-to-end or connection sublayer which sits on top of the hop-by-hop or link sublayer.

The connection suplayer handles *connections* between arbitrary source and destination nodes whereas the link sublayer controls *links* (connections between two neighboring nodes on the route from the source to the destination). The division into sublayers allows to implement control mechanisms on both hop basis and end-to-end basis separately.

On the connection subplayer the data unit is called *segment* and on the link sublayer we call it *fragment*. Of course segments are larger than fragments.

See Figure 2.3 for an illustration of these concepts.

2.2.2 End-to-End Sublayer

This sublayer is running only on the source and destination nodes, not on intermediate nodes. It handles requests from application who want to use the protocol. Upon receiving data from an application it splits the data into segments and passes them down to the hop-by-hop sublayer.

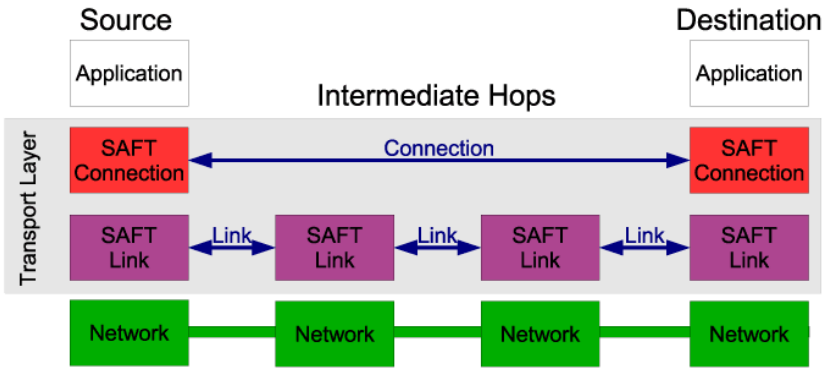


Figure 2.3: Simplified network stack illustrating the terms *connection* and *link* by means of a multi-hop connection.

On the destination node the end-to-end sublayer is simply sorting the data and forwarding it to the application in the right order and sending back acknowledgements to the source. The source node on the other side provides all the control mechanisms.

Unlike TCP there is no mandatory connection establishment procedure. Such a handshake is optional and currently not implemented but might be required by certain appliations.

2.2.3 Hop-by-Hop Sublayer

The hop-by-hop sublayer is running on every node. On the source segments are split into fragments and passed to the *next hop* (the subsequent node on the route to the destination). On the destination fragments are reassembled to segments and delivered to the end-to-end sublayer.

All intermediate nodes buffer and forward all fragments they receive for a limited period of time. The buffering is done for the case that fragments should get lost and need to be retransmitted (here lies the big advantage over TCP on mobile networks, lost fragments are only lost on a local link between two intermediate nodes and not on the whole connection from source to destination).

On every node the hop-by-hop sublayer implements control mechanism for each link. All nodes except the source node also send back acknowledgements to the previous node upon receiving a fragment.

Figure 2.4 illustrates these concepts.

2.3 IP Compartment

For the current SAFT version, the IP compartment is used. Addressing of network nodes is done with IP addresses. The documentation of IP for ANA can be found in [2].

We can query the IP compartment for the address of the next hop for a certain destination. Through the resolve primitive we can ask the IP compartment for an IC to that given address (See Figure 2.5).

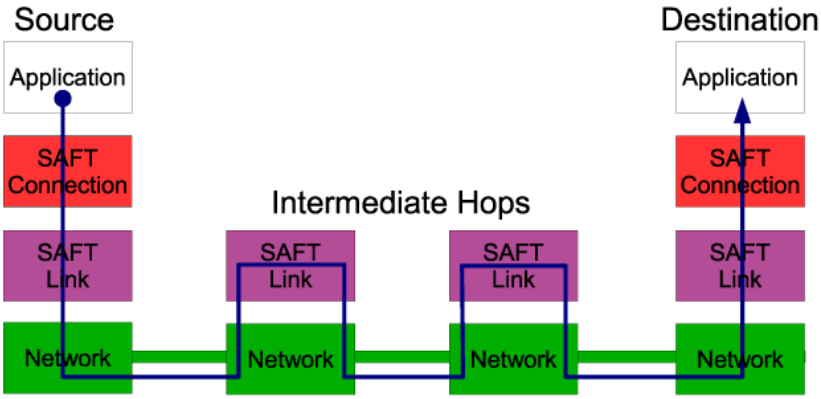


Figure 2.4: Data flow on multi-hop connection.

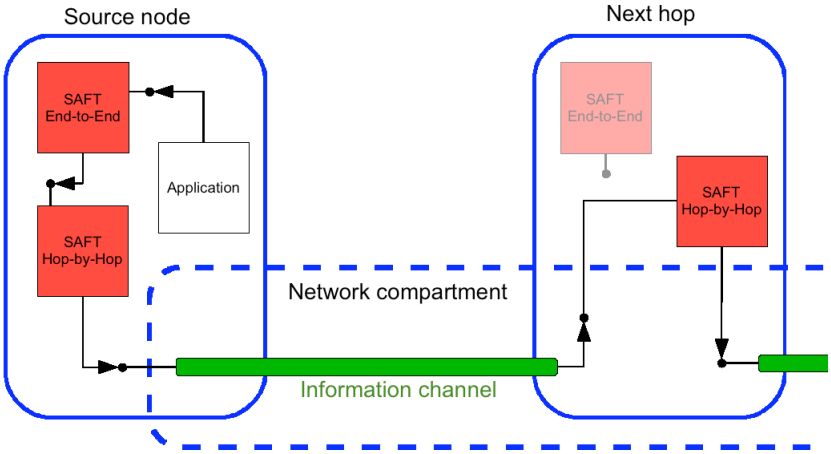


Figure 2.5: The resolve procedure to the IP compartment builds up the IC and returns and IDP which the source's hop-by-hop sublayer is using to transfer data to the next hop. How the SAFT sublayers are divided into bricks and how they resolve each other is described in Section 2.4. *(Image courtesy of Diego Adolf.)*

2.4 SAFT v0.1

Here we will present the important ideas of the predecessor thesis [8]. Note that this first implementation of SAFT on ANA was not a fully reliable transport protocol. It was semi-reliable by providing only the control loops on the hop-by-hop sublayer.

The partitioning of SAFT v0.1 is as can be seen in Figure 2.6. Each sublayer has a main brick responsible for functionalities like sending and receiving. Furthermore they are thought to have one or more control bricks that allow the implementation of control mechanisms which provide reliability (in SAFT v0.1 there is only one existing control brick providing hop-by-hop reliability). That design allows easy switching between several different control mechanisms.

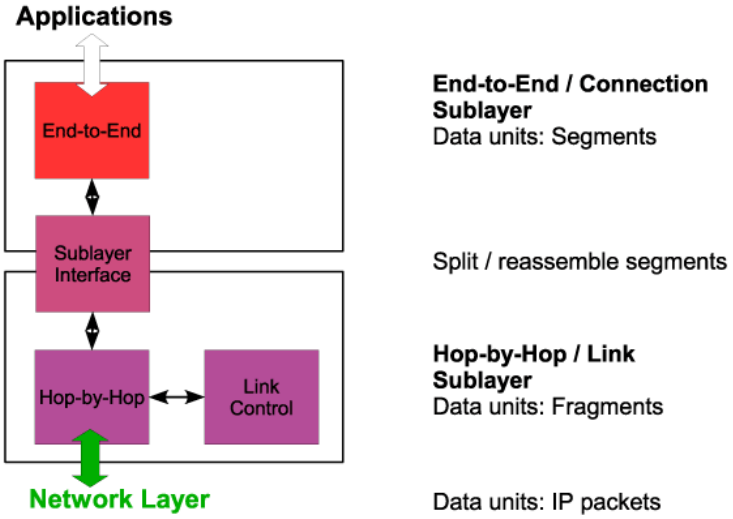


Figure 2.6: Structure of the SAFT v0.1.

There is also the so-called *sublayer interface* (SLI) brick that acts as a mediator between the two sublayers. This allows a better decoupling of the sublayers.

2.4.1 The Bricks

End-to-End Main Brick

This brick is resolvable by applications that want to use the transport protocol. The main functionality is:

- *Sending Data*
  The brick gets the data from the application and encapsulates the data in a SAFT segment (by adding a SAFT header, see Section 2.4.2) and forwards it to the SLI.
- *Receiving Data*
  Incoming segments are passed to the receiving application unless the segment number does not correspond to the number of the next expected segment. In this case the segments got shuffled during the transmission

and need to be reordered before they are forwarded to the application. There is a repository available for storing segments temporarily until it is their turn to be forwarded.

The end-to-end main brick is also the compartment provider brick.

Sublayer Interface

This is the smallest brick with the following tasks:

- Split outgoing segments into fragments and push them down to the hop-by-hop main brick.
- Store incoming fragments in a repository. Reassemble a certain segment if all its fragments are in the repository and forward it to the end-to-end main brick.

Hop-by-Hop Main Brick

This brick handles application data received from the end-to-end main brick over the SLI. Since it is the only brick interacting with the network it does request the next hops and forwards the fragments to the network layer.

- *Store and Forward Procedure*
  The store and forward procedure applies to all fragments that need to be forwarded to another node. These fragments are stored locally in a repository and then they are sent to their corresponding next hop. They are kept in the repository until they get acknowledged.
- *Sending Data*
  Before fragments can be sent, permission is requested from the hop-by-hop congestion control brick. If permission is granted, the sending process can continue. Otherwise, the fragment will be sent later on behalf of the control brick.

  If permission is granted, the IP address of the next hop is fetched from the IP compartment. It 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. Last the IDP to reach the next hop is retrieved from the IP compartment.

  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*).
- *Receiving Data*
  Data is also received in packets. There is a repository helping reordering packets and reassembling fragments.

  Every fragment received is immediately acknowledged to its previous hop (duplicated fragments must also be acknowledged in case a previous acknowledgment was lost). Now depending on the fragments destination it is either forwarded to the next hop or to the end-to-end main brick over the SLI (in case the local node was the destination).

Fragment acknowledgments are also received here, but they are forwarded to the control brick for processing.

Hop-by-Hop Congestion Control Brick

This brick has a repository that saves the state of each fragment (possible states are STATUS_UNSENT, STATUS_ACKED and STATUS_UNACKED) and has a variable that holds the number of unacknowledged fragments.

The brick has the following responsibilities:

- *Granting Sending Permissions*
  It limits the number of unacknowledged fragments on a certain link. 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 has not exceeded the number of unacknowledged fragments, permission is granted. Otherwise, permission is denied and a timer is started which will try to transmit the fragment again in three seconds.

- *Fragment Acknowledgments*
  All fragment acknowledgments are received by the hop-by-hop main brick but are passed to this brick for processing. They trigger an update of the status of the corresponding fragment and of the counter of unacknowledged fragments.

- *Fragment Retransmissions*
  After granting a permission a timer is started which will trigger a retransmission of the fragment after the *retransmission timeout* (RTO) runs out unless a acknowledgment is received within that time duration (the RTO is set to three seconds).

2.4.2 SAFT Header

The preceding thesis also introduced a packet format for SAFT that is also reused in SAFT v0.2 whose header had the following fields. Only fields that are not self-explanatory are explained:

- *source target and context*
  The context can be compared to an address of a node in a network. In our case a context is an IP address. The target is a certain service/application that we want to access on a network node (a target is comparable with a port in IP).

- *destination target and context*
  Has the same meaning as source target and context.

- *type*
  The type of a packet can take 3 different values: data (SAFT_DATA), fragment acknowledgment (SAFT_FRAGACK) and segment acknowledgment (SAFT_SEGACK).

- *segment number*

- *fragment number*

- *packet number*
  The packet number is used for numbering the packets within the network compartment. Most likely the MTU of the network is smaller than the fragment size, the fragments need to split and thus we need to enumerate them.
- *segment acknowledgment number*
  This field is only used if the type is SAFT_SEGACK.
- *fragment acknowledgment number*
  This field is only used if the type is SAFT_FRAGACK.
- *segment length* (in number of fragments)
- *fragment length* (in number of packets)
- *packet length* (in bytes)

Chapter 3

Design and Implementation

This chapter describes the design process involved in this thesis.

First we give a quick overview of what needs improvement in SAFT v0.1 in order to allowed our desired extensions. We continue with a precise description of SAFT v0.2 including all its bricks. Together with the description we also provide flow charts to lighten the understanding of the texts. In the end congestion control, multiple application support and the different message XRP message types are discussed in their own sections.

3.1 Where SAFT v0.1 Needs Improvement

Since SAFT v0.1 was a straight forward implementation with focus on functionality and not on performance or optimization there are several weaknesses. Note that some of the listed issues were not solved in SAFT v0.1 because it would have gone beyond the scope of a single semester thesis.

- In SAFT v0.1 the bricks need to resolve each other according to Figure 3.1 when sending out data. The application resolves and IDP from the end-to-end main brick. Now for every piece of data that the application sends to the IDP the following happens: The data is encapsulated and prepended with a SAFT header, an IDP is resolved from the SLI, the segment is sent to the SLI, the segment data is split into fragments, an IDP is resolved from the hop-by-hop main brick, the fragments are sent to the hop-by-hop main brick, the next hop is resolved, for every fragment an IC is resolved through the IP compartment and then finally the fragments are split to packet and forwarded to the IP forwarding brick (On the receiving side the resolving processes are pretty much vice versa).

  We have loads of resolve and send function calls. As mentioned in Section 2.1.4 using or resolving and IDP will execute a callback function. By the design of ANA many of those functions must be executed as threads to prevent deadlocks. This whole process leads to the creation of a huge number of threads which can quickly cause an overload of the system.

- Fragments that are hold back because the limit of unacknowledged fragments is reached might be hold back even longer since newer fragments

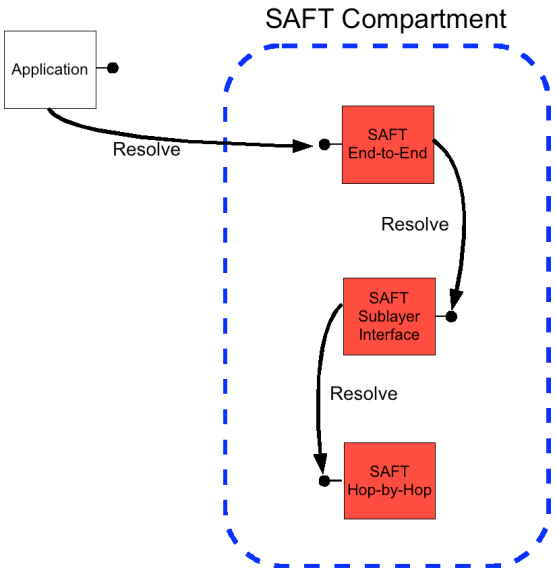


Figure 3.1: Resolve processes for sending. *(Image courtesy of Diego Adolf.)*

might get sent out before the unsent ones. This case occurs when acknowledgments arrive shortly after a fragment was hold back. The acknowledgments open slots for new fragments that can be sent out but since the three second counter for the hold-back fragment only just started it is most likely that newer fragments will be sent during that time.

This should be prevented by checking for hold-back fragments as soon as a slot is opened due to an arrival of an acknowledgment.

3.2 Overview of Structure

The brick structure in Figure 3.2 of the SAFT v0.2 compartment on one node still looks very familiar with version 0.1 (shown in Figure 2.6). The only addition is the *connection control brick*. Most changes are inside the individual bricks.

3.2.1 Resolve/Publish Process at Startup

To eliminate the problem of creating too many threads we tried to statically resolve as many communication channels as possible inside on node at startup. After the launch each brick will know the IDPs over which it can reach the other bricks it needs to communicate with.

No further resolving between bricks on one node will be necessary after this step except for two cases:

- When an application looks for the end-to-end main brick.
- When the IP compartment is asked for an IC to another node.

Of course for both cases it would not possible to resolve at startup.

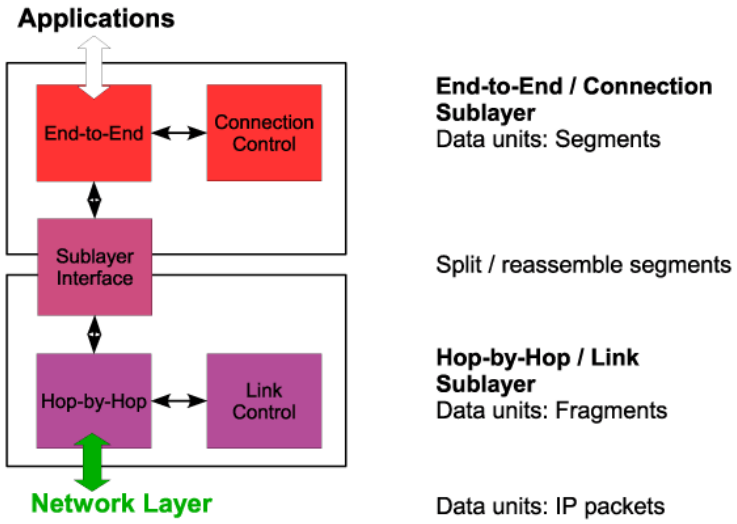


Figure 3.2: Structure of SAFT.

Figure 3.3 shows the startup resolve process. Each of the five bricks publishes itself in the node compartment. The published callback function is called *entrypoint*. In every brick this function is called upon receiving a resolve or a publish request. The entrypoint is responsible for handling control messages (publish, resolve but also internal SAFT control messages that will be described later).

For two bricks that need to get to know each other we always use the same procedure: The first brick resolves the second brick (and gets an IDP on which it can reach the second brick's entrypoint). Now the first brick knows the second. Afterwards the first brick publishes itself in the second since the latter also wants an IDP for accessing the first brick. We could look at this as a handshake between two bricks.

In Figure 3.3 it can also be seen that the handshakes between the end-by-end main brick and the SLI and between the SLI and the hop-by-hop main brick contain an additional resolve. This is not a resolve to the node compartment for finding a brick, it is directly sent to handshake partner (either SLI or the hop-by-hop main brick) requesting another IDP which is only responsible for handling data messages. The entrypoint callback function on the other hand always differentiates the type of message it receives. This is omitted by providing an extra callback for data messages only. From one point of view these additional callbacks could be seen as artifacts from SAFT v0.1 but they also provide a little performance improvement by not sending all the data through the entrypoint. Note that between the main and control brick on each sublayer we always only have control messages and between the main bricks and the SLI we always only have data messages.

Since it is not possible to resolve a brick that has not been started yet the load order of the bricks is important. The hop-by-hop main brick must be started after the link control brick and after the IP compartment is loaded, the SLI must be started after the hop-by-hop main brick and the end-to-end main brick must be started after the connection control brick and the SLI (this can

also be seen in Figure 3.3). This leads to our recommended startup order:

1. everything apart from SAFT
2. link control brick
3. hop-by-hop main brick
4. SLI
5. connection control brick
6. end-to-end main brick
7. applications

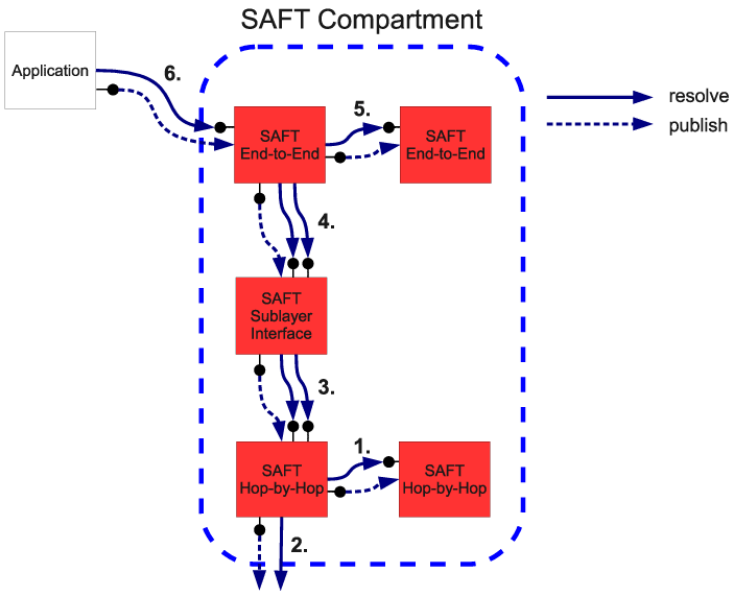


Figure 3.3: Static resolve/publish process at startup. The order of actions is numbered. Note that the publishes to the node itself are not marked. Step 2 symbolizes the handshake between the hop-by-hop main brick and the IP compartment.

3.3 The Bricks

Here is a description of all those bricks whose designs were significantly changed during the thesis (the only brick not described is the SLI, there were changes in the code but the brick's basic responsibilities are still the same as in the SAFT v0.1).

3.3.1 End-to-End Main Brick

The end-to-end main brick has the following responsibilities:

- *Providing ICs to applications*
  The end-to-end main brick is resolvable by the applications and can provide ICs to foreign network nodes. There is a repository that stores IDPs of applications. Note that all applications, also those only receiving data need to resolve the end-to-end main brick. An IC can either be incoming or outgoing.

  There is a second repository that stores information about the ICs (*IC repository*), for each outgoing IC we save the number of the next segment that will be sent out and also the number of the last acknowledged segment. For incoming ICs it is sufficient to know the number of the segment up to which all segments were received.

- *Sending of segments*
  There is the *segment repository* that buffers all the segments. A new segment from an application is put into this repository and a copy of it is appended to a *message queue*. There is a *message handler thread* which polls the message queue and processes the segments one by one in a first in first out order. Processing means that a control message is sent to the connection control brick containing the header of the segment and the segment is forwarded to the SLI. If the queue is empty the message handler thread blocks until it gets notified that a new segment has been appended to the queue.

  The message queue and handler allow to cut down vastly on the amount of created threads (that was one of the big issues of SAFT v0.1). Furthermore this also guarantees that the segments are handled in the right order (in the order they arrived at the end-to-end main brick).

  We appended a copy of the segment to the message queue. It is possible that a segment gets deleted from the repository while it is still waiting in the message queue and therefore we can not use the original segment.

- *Receiving of segments*
  When a new segment is received by the SLI the IC repository is queried for the number of the segment that the associated application expects next. If the incoming segment corresponds to this number it is forwarded to the application. Otherwise if the segment number is higher (meaning that segments got shuffled during the transmission) the segment is buffered in the segment repository until the missing segments are received. In the third case where the segment number is less or equal to the expected segment number we can simply ignore it since this was a retransmitted segment that we already received earlier.

  If the expected segment was received we always also have to check if newer segments were stored in the segment repository. If that is the case those fragments are also forwarded to the associated application. All segments passed to the application are removed from the repository.

- *Segment acknowledgements*
  For every received segment a segment acknowledgement is immediately

sent back to the source node by the end-to-end main brick. In SAFT v0.1 sending of acknowledgements was always ordered by the control brick. There is no need for that, time and an additional control message can be saved be letting the main brick handle it. Note that the acknowledgements are not put into the message queue, they are directly sent to the SLI.

A received segment acknowledgement is forwarded to the connection control brick and the associated segment is deleted from the segment repository.

The segment acknowledgements introduce a control loop on the end-to-end sublayer. This control loop (shown in Figure 3.4) is responsible for the end-to-end reliability.

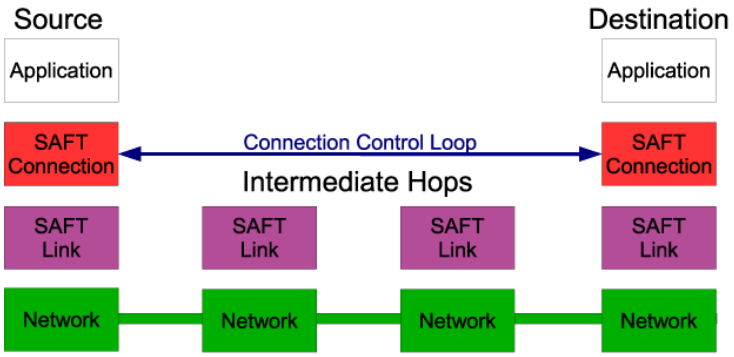


Figure 3.4: End-to-end control loop.

- *Retransmission of segments*
  Sometimes a segment needs to be sent out again (when its acknowledgement did not arrive in time). This is not a problem since all the segments are stored in the segment repository. Retransmissions are always ordered by the connection control brick through a control message. When a retransmission order arrived after the segment has already been deleted from the repository it can simply be ignored (because it was deleted due to the arrival of its acknowledgment what renders the retransmission redundant).

3.3.2 Connection Control Brick

The connection control brick has the following responsibilities:

- *Update Statistics*
  The connection control brick gathers a lot of data. It always gets informed either when a segment is sent out or when a segment acknowledgment arrived. The brick has two repositories: the *segment status repository* and another repository that saves information about roundtrip times. The latter one saves all the information that is needed to dynamically update the RTOs for congestion control (see next section). This repository has information about every destination node SAFT ever sent data to.

- *Retransmissions*
  When a segment is sent out the exact time of that moment is saved in

the status repository. When the according acknowledgement arrives the time is measured again. The difference is the RTT for that segment which helps setting appropriate RTOs (see Section 3.5).

Furthermore two timers are started when a segment is sent: One with the RTO that would trigger a retransmission if the acknowledgement for that segment did not arrive in time. In case of a retransmission this timer is restarted. The other timer is an error timeout that is set to two minutes. If no acknowledgement for that segment is received within those two minutes the transmission will be aborted and an error will be displayed. (In the current state this error is just written to the end-to-end main brick's log file, in future a error message should be send to the corresponding application).

When an acknowledgement arrives the corresponding entry is deleted from the status repository and the two timers are stopped.

3.3.3 Hop-by-hop Main Brick

The hop-by-hop main brick works similar to the end-to-end main brick but is a little more complicated. This brick does not know about different applications or clients that use the protocol. All it deals with are its direct neighbors in the network.

The responsibilities of this brick are:

- *Sending fragments*
  There is also a *fragment repository* like the main brick of the end-to-end sublayer has its segment repository.

  The probably biggest difference is that on this sublayer there are three different message queues: one for acknowledgements, one for retransmissions and one for normal fragments. The idea is that each queue has a different priority. The acknowledgements queue has the highest priority, then comes the retransmission queue and with lowest priority there is the normal message queue. A message handler thread, like the one in the end-to-end sublayer, is polling and processing the items in the queues. Queues with higher priorities are favored and are handled before the others.

  The other important difference is that data fragments are not simply appended to one of the message queues but first the hop-by-hop control brick has to be asked for permission. So new data fragments that want to be sent out are first put into the fragment repository after entering the hop-by-hop main brick. This is true for fragments that come from the local node but also for fragments received from other nodes that need to be forwarded. The only expections are fragments destined for the local node, they are not buffered but forwarded upwards to the SLI.

  The Second step is to send a permission request to the control brick. Depending on the response the fragment is either appended to the according queue or nothing is done. In the latter case it is the control bricks responsibility to order the transmission of that fragment later on.

- *Receiving fragments*
  Upon receipt of a fragment there are two things to distinguish: Is the

fragment for this node? What type of fragment is it (data, fragment acknowledgement, segment acknowledgment)? Data fragments are either forwarded by going through the procedure described above or reached their destination. See under acknowledgements to see how the acknowledgments destined for this node are handled. Segment acknowledgement that need to be forwarded are appended to the acknowledgement queue.

- *Retransmissions*
  Work the same way as in the end-to-end sublayer. Fragments that are retransmitted have a higher priority than normal data and are therefore appended to the retransmission queue.

- *Acknowledgements*
  The sending and receiving of fragment acknowledgements works the same as in the end-to-end sublayer. Received segment acknowledgements are forwarded to the SLI which in turn will forward them to the end-to-end main brick.

  Note that the hop-by-hop main brick also interprets segment acknowledgements. A segment acknowledgement can also be seen as a cumulative acknowledgement for all fragments in that segment. So all the fragments still stored in the fragment repository that are part of the acknowledged segment are purged.

- *Communication with the IP compartment* As mentioned in Section 2.1.2 the fragments are split into IP packets before they can be processed by the IP compartment. Vice versa the packets need to be reassembled to fragments upon arrival. This is similar to the split of segments and fragments handled in the SLI and was already present in SAFT v0.1.

  When sending data we always know the destination IP address. But we can't directly send the fragment to the destination if there are intermediate hops between our node and the destination. The IP address of the next hop on the route to the destination can be fetched through a query to the IP compartment. Implementation wise this query to the IP compartment is a simple call to the ANA function *anaL2_requestReply* which prompts the *ip_fwd* brick for an immediate answer to a certain request message. In our case this request message only contains the destination IP. The ip_fwd brick will look up the IP address of the next hop (which is the next directly connected node on the route to the destination) in its forwarding table. This IP adress is put into the answer message that is sent back to the hop-by-hop main brick.

  Now we know the IP address of the next hop but we need an IDP for it in order to communicate with it. The IDP of the next hop can be retrieved through a resolve to the *ip_enc* brick.

  The assignment of IP addresses to IDPs does not change, therefore these mappings can be stored in a repository. But it is possible that the IP address of the next hop changes (on mobile networks link failures and motion of nodes can quickly result in different routing) so it has to be requested for every fragment. It is thinkable to cache the next hop addresses for a few seconds to save CPU time.

3.3.4 Link Control Brick

This brick does not need a lot of explanation since it almost works like the connection control brick. Instead of gathering data about routes to different destination this brick only collects information about its neighborhood. The neighborhood is understood as all network nodes that are directly connected to this node. The information is stored in a repository. Each neighbor that we have data about has an entry in that repository. Such an entry knows the delays to that neighbor, has a list that stores which fragments have not been sent out yet (the fragments that were not granted sending permission) and knows the number of unacknowledged fragments that are on the link to the neighbor.

Furthermore this brick also sends the permission responses. Either the sending is granted or it is denied. If it is denied this brick will send a control message later on to the hop-by-hop main brick. This control message will order the sending of the fragment that was hold back. More detailed explanations about when the sending is granted/denied and under what circumstances a hold-back message is ordered to be sent can be read in Section 3.5.

3.4 Flow Charts

To illustrate the chain of actions described in this chapter we also include two simplified flow charts. A graphical representation might simplify the understanding.

The end-to-end sublayer is less complicated than the hop-by-hop sublayer and it works similar. Therefore we abstain from including the end-to-end sublayer in the flow charts. If you have understood these charts you will also understand how the end-to-end sublayer works.

3.4.1 Sending Data

Figure 3.5 explains what happens if a fragment coming from the SLI arrives at he hop-by-hop sublayer and wants to be sent out. The steps are enumerated with black numbers. Note that solid arrows visualize data messages whereas dotted arrows are control messages. The red lines show the ways of acknowledgements. This happens after the original seven sending steps. All shown components are part of the hop-by-hop main brick except the *Controller* in the upper right is the hop-by-hop control brick.

Important is also that the data messages are not sent or forwarded to the message handler (step 6). The message handler itself is responsible for taking the messages out of the queues.

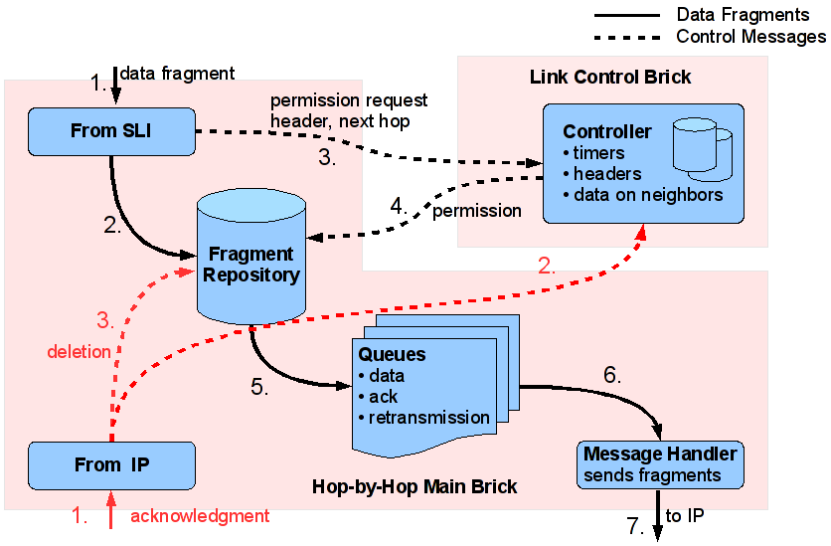


Figure 3.5: Sending data.

3.4.2 Receiving Data

Figure 3.6 explains what happens when a fragment coming from the network compartment is received by the hop-by-hop main brick. Red lines again show the ways of acknowledgements. This happens for every incoming fragment.

For data fragments the way in the chart depends on wether the node is the destination. If that is the case the data is forwarded to the SLI (arrow on the left, marked as step 2). Otherwise the forwarding process is ignited: the fragment is put into the fragment repository (second step 2 arrow) and the controller brick is asked for permission (step 3), the rest works the same way as for sending data. Just follow the arrows.

3.5 Congestion Control

3.5.1 Limiting Unacknowledged Data

Congestion is avoided by limiting the number of unacknowledged segments or fragments. The hop-by-hop sublayer simply does that whereas the end-to-end sublayer uses a sliding window (which is a little improvement over just limiting the number of unacknowledged segments). With the sliding window a new segment with number x can only be sent if the segment with number $x - window\_size$ was acknowledged. Limiting the number of unacknowledged segments rather just provides a number of sending slots, a slot is considered as occupied as long as the acknowledgement has not arrived.

On the hop-by-hop sublayer it would not be possible to use a sliding windows because the fragments that go over one link belong to different applications and there is not a link specific numbering of fragments. For using a sliding window here one would have to reenumerate all the fragments going over a link (and redo this on the other side of the link, this would lead to another encapsulation

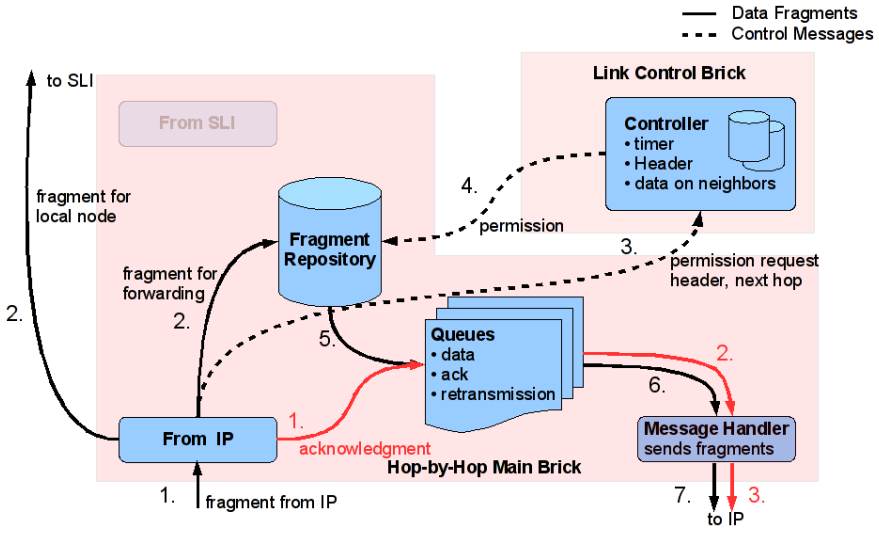


Figure 3.6: Receiving data.

of the messages).

Fragments are not retransmitted infinitely many times. There is also a limit (set to ten in the implementation) that says how many times a fragment is retransmitted at most. If that is not enough the only hope is that the next segment retransmission will be successful.

If fragments have been detained because the limit of unacknowledged fragments was reached and a fragment acknowledgement arrives a slot for sending data fragments becomes open. Since the link control brick has a list of yet unsent fragments it can now order the hop-by-hop main brick to send the fragment that is next in row.

3.5.2 Retransmission Timeouts

On both sublayers the new RTOs are calculated with Jacobson's algorithm (see next Section). On the end-to-end sublayer we use exponential backoff to increase the RTOs in case a segment didn't get acknowledged several times in a row. This would not make sense for fragments since the timeouts there are not due to congestion but rather due to link failure.

Jacobson's Algorithm

For the very first segment sent on a new connection (or a fragment sent to a before unknown neighbor node) the RTO is set to an initial value. All further segments/fragments will dynamically adjust their RTOs based on past RTT samples. The new RTOs are calculated with Jacobson's algorithm [14]. The RTT for the next segment/fragment is estimated and this estimate is used again to determine a suitable RTO. The algorithm works as follows:

First we calculate the difference between the sample and the estimation.

$$\texttt{Difference} = \texttt{SampleRTT} - \texttt{EstimatedRTT} \tag{3.1}$$

This difference is used to set a new estimation.

$$\texttt{EstimatedRTT} = \texttt{EstimatedRTT} + (\delta \times \texttt{Difference}) \tag{3.2}$$

The next step updates the variation.

$$\texttt{Deviation} = \texttt{Deviation} + \delta \times (|\texttt{Difference}| - \texttt{Deviation}) \tag{3.3}$$

Then, finally we can produce a new RTO as weighted average of `EstimatedRTT` and `Deviation`:

$$\texttt{RTO} = \mu \times \texttt{EstimatedRTT} + \phi \times \texttt{Deviation} \tag{3.4}$$

The parameters δ, μ and ϕ need to be set according to the target network. Usual values are $\delta = 1/8$, $\mu = 1$ and $\phi = 4$ [15] (this are also the values used in our implementation).

Note that the first estimate and also the deviation have to be set to initial values. For the deviation this is simply zero.

3.5.3 Throttling of Applications

For throttling applications that send too fast we use a similar solution as the UNIX sockets provide. The return value when sending data over UNIX sockets says how much of the data was accepted to send. If not all is accepted the user is responsible to try resending the rest a short time later. Besides the simple send function (that sends a bunch of data do an IDP) there is also a higher level function that does not only send but that requests a reply message (the already mentioned anaL2_requestReply function). Applications have to use this function. The reply message always reports if the data was accepted or if the system is used to capacity. Right now we only differentiate only between accepted or not accepted, in future it would be nice to also accept only parts of the data. Note that if the data is accepted it does not mean that the data successfully arrived at the destination. It only means that there is enough capacity to try to send the data. The data is accepted if the sliding window is not full.

3.6 Multiple Application Support and Identifiers

Since several applications might send segments/fragments with the same number at the same time it is important to have a mechanism to distinguish those. All information about segments/fragments is always saved in repositories. The question is what keywords should be used to address the entries in the repositories. SAFT v0.1 just used the segment numbers on the end-to-end sublayer and a segment and fragment number concatenated for the the hop-by-hop sublayer.

In our case the segments are identified in the repositories by a concatenation of the source address, the source target (source application) and the segment number. The three expressions form an identifier string.

For the hop-by-hop sublayer it works similar but in addition we also include the the destination address and the fragment number into the string to keep identifiers in the repositories unique.

The hop-by-hop sublayer does not care at all about different applications. It bases all its decision only on the next hop address and the states of the links to its neighbors. Differentiation between the application is only done to prevent confusing between fragments with the same numbers.

The end-to-end sublayer allows to be resolved by an arbitrary number of client applications that will all append there data messages to the same queue. Those who get their acknowledgements back fast will also be able to send faster. If the acknowledgements don't arrive quick enough the application might be rejected the next time it wants to send data of the resolved IDP.

As mentioned before the limiting is done using a sliding window on the end-to-end sublayer. Right now there is only one global window available. It would make more sense to have a separate window for each IC/application. This option was not implemented due to lack of time.

3.7 Message Types

ANA allows users to define their own XRP commands. We introduce only the command types that were added in this thesis (all the remaining control messages used are described in the preceding thesis [8]):

- *XRP_CMD_SENTSEG*
  With this message the end-to-end main brick informs the end-to-end control brick that a segment was sent out. The message includes the header of the sent segment.

- *XRP_CMD_DATA_REPLY*
  This message type is received by the applications when they forward their data to the end-to-end main brick. Such messages represent the feedback to applications. There is only one field, if it contains 0 the data was rejected and 1 means the data was accepted. In future one could think about expanding it and also accepting data partially (as it is the case with UNIX sockets).

- *XRP_CMD_RESETFRAG*
  The hop-by-hop main brick sends this message to the corresponding control brick when the status of a fragment shall be resetted. This happens when a segment is retransmitted. All status' of the associated fragments need to be resetted. When a segment is retransmitted its fragments are treated like new fragments (timers are reset and the number of retries is set back to zero).

  This message type has one field containing the id of the corresponding fragment.

- *XRP_CMD_FIRSTSEND*
  This type is sent by the end-to-end control brick to the end-to-end main brick to order the sending of a fragment that was hold back. We can't use the retransmission type XRP_RESEND since fragments sent for the first time are handled different than retransmissions.

  This message type also has one field containing the id of the corresponding fragment.

Figure 3.7 shows the source and destination bricks four those four new XRP messages types.

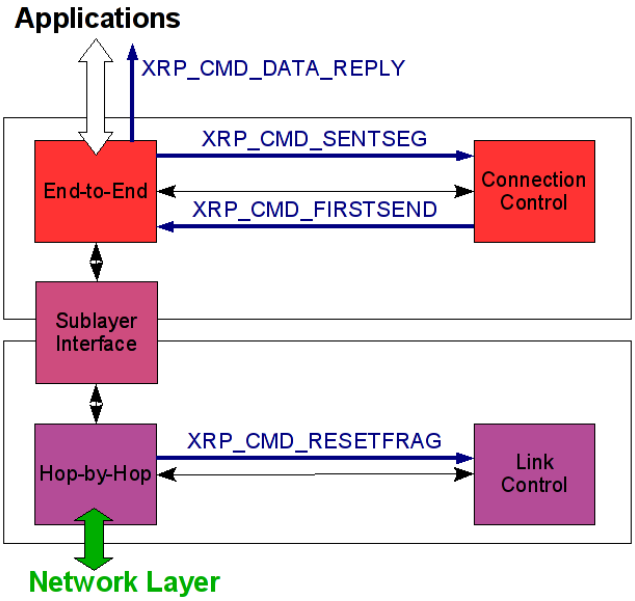


Figure 3.7: XRP command types introduced in SAFT v0.2.

Chapter 4

Validation & Evaluation

4.1 Validation

A validation was performed to test the functionalities in the SAFT compartment and to assure the correctness of the protocol. For testing the *saftDemo* brick was used. It allows to send arbitrary files from a source to a destination. It is also possible to have several transmissions concurrently. This demo brick can be understood as a normal application that wants to use the SAFT protocol.

The Linux tool *netem* allows to test the protocol's behavior in many different cases: What happens when packets get lost (packet loss) on a certain link? What happens when packets get duplicated or what is the outcome if packets are reordered during transmission.

4.1.1 Test Scenarios

SAFT had to prove itself on different scenarios (Figure 4.1.1).

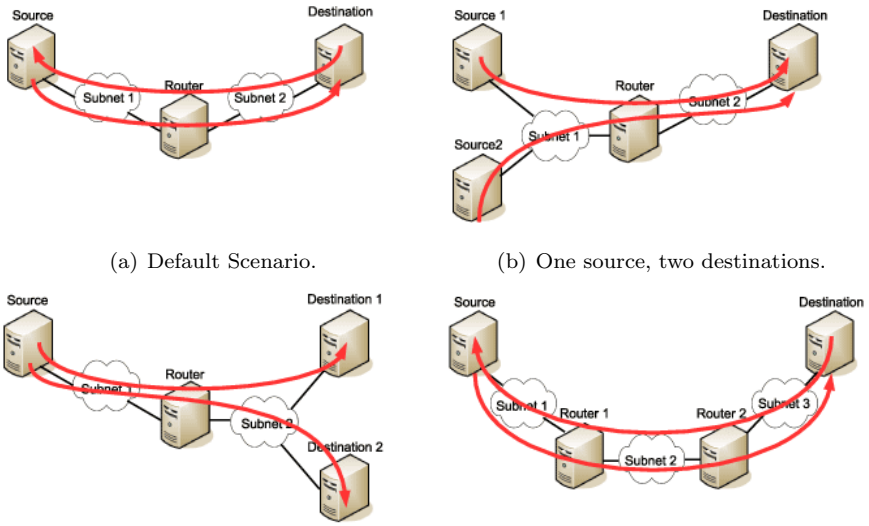
The scenario (a) is the simplest test case with two end nodes and one intermediate router. One end node sends data to the other end. Concurrently the receiving node can also transmit data to the other side (and the packets cross on the router).

The scenario (b) shows the case where two source nodes send data over the router to a destination at the same time.

The bottom left scenario (c) shows the opposite: a single source is sending to two destination concurrently.

The last scenario (d) considers the case where data is transmitted over more than one intermediate node.

All those scenarioes can also be executed with more than one connection between each source and end node. Together with netem these scenarios allow a solid validation. The different netem settings (packet loss, duplicates, reordering) were systematically checked for the scenario in Figure (a) with several runs. For the remaining scenarios we did only random samples for the three netem cases.



(a) Default Scenario.

(b) One source, two destinations.

(c) Two sources, one destination.

(d) Two intermediate hops.

Figure 4.1: Validation Scenarios.

4.2 Encountered Problems

Unfortunately during the testing phase we encountered several weaknesses and issues with the ANA framework. Many of the (*resolve*) and *request_reply* calls time out because the response does not arrive in time under high load.

This happens especially often when those functions are called repeatedly without pausing in-between. There are many loops in the SAFT implementation that contain calls to the mentioned functions (sending of data in the demo application, splitting of segments, splitting of fragments, the two message handler loops, ...). We were forced to artificially slow down the implementation by inserting sleeps inside these loops to keep the SAFT protocol stable. The forced sleeps last from 10msec up to 250msec.

4.3 Evaluation

The encountered problems make it difficult to perform a meaningful evaluation. Nevertheless we measured the performance of SAFT for the upper left scenario in Figure 4.1.1. A single file containing random data (30KB big) was transmitted from the source to the destination. The test was executed for different levels of packet loss (0%, 4%, 8%, 12%, 16%, 20%, 24%, 28%). For each level the test was run at least 50 times. The results are shown as red columns and the performance of SAFT v0.1 is also displayed in blue (with packet loss levels 0%, 5%, 10%, 20%).

The machines used for the test had the following properties: Intel Pentium 4 CPU 1.80GHz, 512MB ram and 10/100Mbps Ethernet device with Debian Linux 4.0 as operating system.

As can be seen the performance improved significantly over the old version. However if you consider that without packet loss the throughput is only 15KB/s

(2sec for one transmission) the results are not too good. This still low performance can be blamed on all the sleeps that were included in the code (see last section). Furthermore the ANA framework has never undergone a performance evaluation and is therefore not optimized at all.

Still, the relative performance is not too bad. With 28% packet loss it takes only 3.5 times longer to transmit one file (2sec without packet loss and 7sec with 28% packet loss).

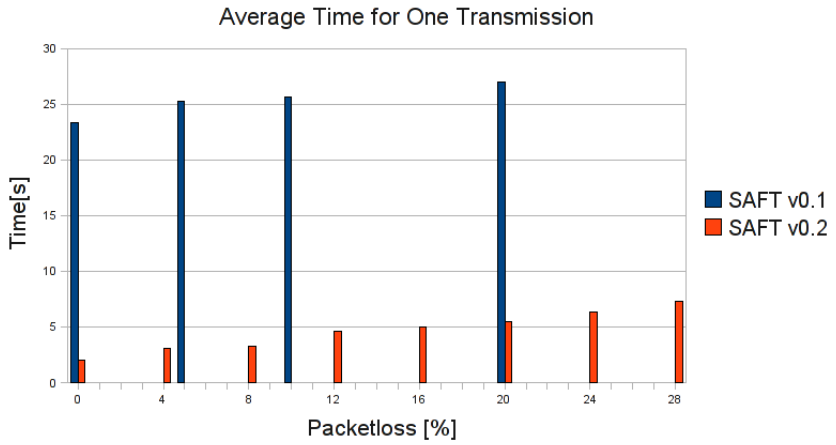


Figure 4.2: Evaluation: 50 x 30KB Transmission.

When taking out all the sleeps mentioned in the last section a performance increase of almost factor 10 could be experienced. But unfortunately this lead to a totally unstable situation where only a small fraction of the transmission succeeds. Tests with bigger file sizes failed for the same reason despite the included sleeps. The occurence of only one timeout suffices fail the transmission.

Chapter 5

Conclusion

With this semester thesis, the already existing implementation of the SAFT protocol was enhanced from SAFT v0.1 to SAFT v0.2. The before semi-reliable protocol is now fully reliable and earns the name of a transport protocol. Therewith SAFT is the first transport protocol for ANA.

All the bricks were design in a flexible and modular fashion that allows their reuse. We hope with this precursor protocol for ANA actuates and motivates the implementation of other protocol for ANA. SAFT is supposed to be a good example and foundation for new ANA protocols.

SAFT of course is not perfect yet and Section 5.2 gives new ideas that could be explored. Nonetheless the validation and evaluation have demonstrated that SAFT works correctly and that the most important control mechanisms for congestion control are available.

We can also say that it is a very difficult task to separate a transport protocols functionality into several bricks because all the parts strongly depend on each other. The tradeoffs of having this modular design are the loss of performance and the more complicated implementation.

Unfortunately also some negative issues were highlighted during this thesis: We saw that instabilities and timeouts can occur when transmitting data too fast between bricks. ANA makes it also very easy and tempting to overload the system with a great many of threads. The former issue led to the fact that SAFT was throttled artificially to keep it stable and not to crash. Currently the performance bottleneck of SAFT lays within the ANA framework.

5.1 Contributions

Throughout the thesis following contributions were adduced:

- SAFT for ANA was ported from ANA framework version 0.1 to version 0.2.
- End-to-end reliability was added through the sending of segment acknowledgements.
- Congestion control was achieved on both sublayers by:
  - Limiting the number of unacknowledged segments/fragments.

  - Dynamically calculating the RTOs for retransmissions.
  - Implementing exponential backoff for retransmissions on end-to-end sublayer.

- Support for multiple applications was added. This allows several programs to use the protocol concurrently.
- The performance was increased through redesign of the existing bricks.
  - Message queues and message handlers were implemented to prevent the creation of too many threads.

5.2 Future Work

There are several issues that could be improved through further research or implementation:

- *Variable Network Compartment*
  Currently SAFT only works with IP as the network compartment. A more generic solution which does not link to a certain compartment would be more in favor of the ANA idea and would allow better reusability.
- *Performance Evaluation*
  A performance evaluation would certainly help to determine SAFT's current capabilities and would also show the bottlenecks that need optimizing.
- *Performance Comparison with TCP*
  Using existing bricks would allow to implement a ANA-TCP and would provide ways to practically prove SAFT's superiority over TCP on wireless networks but also that it performs worse on wired networks.
- *Support for Kernel Module*
  Make it possible to run the SAFT implementation as kernel module (ANA already provides ways to load brick as kernel modules).
- *Flow Control*
  Creation of another control brick inside the hop-by-hop sublayer which informs the sender about the buffer state of the receiver and throttles the sending rate if necessary to prevent buffer overflows
- *Dynamic Reconfiguration of the Protocols Parameters*
  Most of the parameters (segment/fragment size, initial values for Jacobson's algorithm, sliding window size and number of unacknowledged fragments) are currently hard coded. In a first step we could find the optimal static values for those variables. In a second step we could try to readjust them in real time under the influence of the environment.
- *Fair Sharing of Link Between Applications*
  The sharing is currently only done by preventing applications who don't get their acknowledgments soon enough from sending and the rest is handled by the operating system scheduler. The scheduler decides which

thread is running next and which application will be allowed to enqueue the next bunch of data in the corresponding queue.

Sharing could be made fairer by measuring the sending rates and throughputs on the links and connections and allow access to sending on basis of those measurements.

Appendix A

Implementation Details

This chapter gives a small overview over the most important data structures, message types and parameters.

Note that only the most important parts of the implementation are covered. It is meant to facilitate the access to the code. If you consider to expand the SAFT v0.2 there is no way around reading the doxygen documentation and the code itself.

A.1 Important Data Structures

The most importante used data structures are (only not self-explanatory fields are explained):

A.1.1 Status

```
struct status {
        uint8_t retries;
        uint8_t status;
        unsigned int timerId;
        unsinged int errTimerId;
        unsinged int rto;
        struct timeval sendTime;
        char *nhop;
};
```

This struct is used in repositories of the control bricks to store information about segments/fragments. *retries* specifies how many times the segment/fragment was retransmitted. The *status* can be STATUS_UNSENT, STATUS_UNACKED or STATUS_ACKED. The timer identifiers are used for the timeouts, the first for the RTO and the second one for the error timeout (not used by the hop-by-hop control brick because it is not necessary since fragments are only transmitted a limited number of time). *sentTime* is the exact time when the segment/fragment was sent out. *nhop* is the IP address of the next hop.

A.1.2 Message Holder

In all the repositories or in the message queues we never save only the serialized SAFT messages. We always use so-called *message holders*:

```
struct msgHolder {
        void *data;
        int len;
        struct saftHdr hdr;
        char *nhop;
        int retries;
};
```

data points to the serialized message, *len* is its length. The SAFT header *hdr* is present because we don't want to deserialize the message every time we need to access a field in the message's header. This struct is only used in the two main bricks whereas the status struct above only emerges in the control bricks. Since sometimes the control brick also needs to know how many times the segment/fragment was retransmitted the last field *retries* also occurs here (to save two messages between the main and control brick).

A.1.3 Roundtrip Information

This struct holds all the information that is required for Jacobson's algorithm to dynamically upgrade RTOs.

```
struct rtInfo {
        int estimatedRTT;
        int RTO;
        int deviation;
};
```

A.1.4 Neighborhood

```
struct neighbor {
        struct rtInfo rt;
        int unackedFrags;
        int unsentFrags;
        struct unsentFrag *firstOfUnsent;
};

struct unsentFrag {
        struct unsentFrag *next;
        char *id;
};
```

Used by the hop-by-hop control brick. The latter struct is a linked list of fragments that were hold back and that need to be sent out as soon as sending slots open up. The first struct *neighbor* has the roundtrip information to that neighbor *rt*, the next two fields save how many unacknowledged fragments that are on the link and how many messages are still hold back. The last field is the starting point of the linked list of unsent fragments.

A.1.5 IC Statistics

```
struct icStat {
        uint16_t nextSeg;
        uint16_t lastAckedSeg;
};
```

This struct is used by the end-to-end main brick to store for every outgoing IC the number of the next segment that will be sent and also the number of the last acknowledged segment. For incoming ICs only the *nextSeg* field is used. It presents the number of the next expected segment.

A.2 Used Parameters

Here is a list of the current SAFT parameters (no big effort was undertaken so far to optimize these parameters). In the brackets we show the name used in the code. All parameters are defined inside saft.h with exception of the values for Jaconbson's algorithm (those are set independently for both subplayers in saftCCC.c and saftLCC.c).

- application MTU (MTU_APP): 10140B
- fragment size (MAX_SAFTFRAG_SIZE): 2560B
- IP packet size (MAX_SAFTPKT_SIZE): 640B
- sliding window size for segments (SEG_WINDOW_SIZE): 4
- initial segment RTO (DEFAULT_SEG_RTO): 10sec
- initial assumed segment RTT (START_SEG_RTT): 0.5sec
- segment error timeout (SEG_ERR_TIMEOUT): 120sec
- max. number of unacknowledged fragmentson one link (MAX_UNACKED_FRAGS): 7
- max. fragment retransmissions (MAX_FRAG_RETRIES): 10
- initial fragment RTO (DEFAULT_FRAG_RTO): 1.5sec
- initial assumed fragment RTT (START_FRAG_RTT): 0.2sec

- Jacobson (note that we currently use the same values on both sublayers):
  - End-to-end sublayer
    - δ (JAC_ETE_DELTA): 1/8
    - μ (JAC_ETE_MU): 1
    - ϕ (JAC_ETE_PHI): 4
  - Hop-by-hop sublayer
    - δ (JAC_HBH_DELTA): 1/8
    - μ (JAC_HBH_MU): 1
    - ϕ (JAC_HBH_PHI): 4

Appendix B

How to

Here we provide a tutorial that shows how to install and run SAFT and all the additional software. For developing and testing it is the easiest to install virtual machines on one physical host instead of using several real machines. The use of real machines makes it more difficult to propagate and test code changes.

We recommend to use three virtual machines, two end nodes and one intermediate router connected like in Figure B.1.

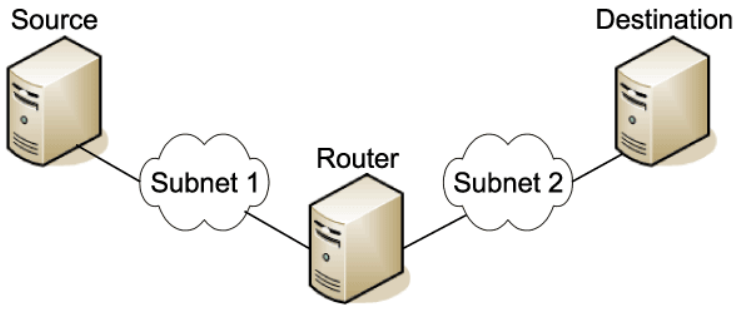


Figure B.1: Three node test scenario.

In this tutorial we suggest *Virtual Box* [16] as virtual machine. The next section specifically explains how to set up Virtual Box for the need of ANA and SAFT. Feel free to use any other virtualization software like VMWare [17], we just recommend Virtual Box because it is open source like SAFT and ANA themselves.

In this thesis the operating system used on the virtual machines was a *Minimal Ubuntu* [18] that provides only the most necessary components and a command line interface. The software should run on every Linux distribution though. Again feel free to use your distribution of choice. It might be possible that some of the settings described in this tutorial might defer if using another distribution.

B.1 Installation of Virtual Box

Install Virtual Box through your distribution's packaging system and download a CD image for Minimal Ubuntu from `https://help.ubuntu.com/community/Installation/MinimalCD`. Then use this image to set up the first virtual machine. Create at least two Ethernet devices:

- The first one is set to a virtual network with a name of your choice.
- The second one is set to NAT and will allow to access the Internet from within the virtual machine over the host machine (this is for downloading ANA, SAFT and every other additional software).

B.2 Set up Shared Development Folder

We will use a so-called shared folder which can be accessed by both, the three virtual machines and also the host operating system.

To use shared folders within Virtual Box you will need the *Virtual Box Guest Additions*. If you are lucky clicking through Devices ⇒ Install Guest Additions will do the whole magic. If you should experience problems you will find help in Section 4.3 of the Virtual Box manual that can be found on `http://www.virtualbox.org/wiki/Downloads` (URL last accessed on 2009-2-18).

It's possible that you will be required to install a kernel module to get the Guest Additions running. If necessary Virtual Box will tell you the exact name of the software package that you can download over your distributions package system.

A shared folder can be mounted by the following command:

```
$mount -t vboxsf sharename mountpoint
```

Where sharename is the path of the shared folder on your host system and mountpoint is the name of the share within the virtual machine. We recommend to add the mount command to your /etc/rc.local file so that the share is always mounted at boot time.

B.3 Installation of ANA and SAFT

Please assure that you've read the documentation on how to setup ANA [9] before continuing reading. Use subversion to checkout the devel branch from the ANA repository. Add the keyword saft to USER_PLUGIN_BRICKS in the config.txt.

With the current version it is necessary to run ANA and SAFT as root. We recommend to set to set the ANA_BASE_DIR environment variable directly inside the file /root/.profile. Simply add the line

```
export ANA_BASE_DIR=path_to_your_shared_folder
```

to that file.

Before you will be able to launch ANA and SAFT on the virtual machines you also need to update /etc/ld.so.conf file. Append the path of the lib directory inside your devel directory to bottom of the file. Then execute:

```
$ldconfig
```

This makes sure that our shared libraries will be found when launching the application.

B.4 Cloning a Virtual Box

Since we don't want to repeat the whole process for another two virtual machines we can also clone the existing one:

```
$cd dir_that_contains_virtual_machine
$VBoxManage clonevdi machine1.vdi machine1clone.vdi
```

On a cloned Ubuntu the Ethernet devices usually will only work after you delete the file /etc/udev/rules.d/70-persistent-net.rules and after rebooting the virtual machine. This file stores static information about your ethernet cards and some of the cards parameters (like the MAC address) will defer on the cloned machine.

B.5 Running the Demo Script

In the folder devel/C/bricks/scripts/ there are scripts to load ANA and all required bricks. Copy int.sh and end.sh into the devel folder. These scripts are used to build the setup from Figure B.1. Int.sh is used for the intermediate node, end.sh for the two end nodes.

The two script work as follows:

```
$./int.sh vlink_id1 interface1 ip1 netmask1 \
          vlink_id2 interface2 ip2 netmask2 \

$./end.sh vlink_id interface ip netmask gateway
```

Please read the documentation [9] if you don't know what the *vlink_id* is. *interface* is the ethernet interface used, *ip*, *netmask* and *gateway* are the node's IP address, its netmask and gateway. Of course the intermediate node needs double the parameters (except for the gateway) since it uses two separate (virtual) ethernet interfaces.

Or as a concrete example. The intermediate node should be launched first:

```
$./int.sh 1 eth0 10.0.1.1 255.255.255.0 \
          2 eth0 10.0.2.1 255.255.255.0
```

Start node:

```
$./end.sh 1 eth0 10.0.1.2 10.0.1.1 255.255.255.0
```

End node:

```
$./end.sh 2 eth0 10.0.2.2 10.0.2.1 255.255.255.0
```

Then to load the saftDemo brick use:

```
$/bin/mxconfig load brick saftDemo num_receiving_services \
          name_recv_srvc1 name_recv_srvc2 ... \
          dest_ip1 file1 dest_ip2 file2 ...
```

This demo brick provides the possibility to open several sending and receiving ICs. *num_receiving_services* specifies how many receiving ICs we want. The names of the receiving services follow (there need to be exactly as many names as there are receiving services).

Next come the sending services with pairs of destination IP address and the path to the file which shall be sent. These services will be automatically named $saftDemo1$, $saftDemo2$, etc..

Here an example to send two files from the source to the destination). On the destination:

```
$/bin/mxconfig load brick saftDemo 2 saftDemo1 saftDemo2
```

And on the source:

```
$/bin/mxconfig load brick saftDemo 0 10.0.2.2 file1 \
                                     10.0.2.2 file2
```

The files received on the end node will be stored in ~/saftDemoRecv/. If you want to send files from both end nodes to the other end node at the same time (data crossing on the intermediate node) you will have to load the saftDemo bricks on both end nodes almost immediately. That is because they both want to build an IC the other end and this is only possible if both sides are up. However we included a one second delay inside the saftDemo brick to make such tests possible. One second should be enough to enter the load command on both ends.

Appendix C

Problem Statement

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis

Implementation of SAFT on ANA II

Stefan Lienhard

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

September 2008 - December 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.

In a recent semester thesis we have implemented a basic version of SAFT in ANA. The goal of this semester thesis is to further improve the framework and to implement and compare different control mechanisms for the SAFT protocol.

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 on ANA.

We will improve the available SAFT on ANA implementation in several steps: Support multiple applications on top of SAFT, add and evaluate control mechanisms for the SAFT implementation. Furthermore we will explore how we can build other transport protocols with the available building blocks (ANA-UDP, ANA-TCP etc.) 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].
- Study the literature on SAFT on ANA as soon as it is available [5].
- Familiarize yourself with the ana svn and the ana wiki [7].
- Setup a Linux machine on which you want to do your implementation.
- Setup a test network (either physical or virtual) that runs the available SAFT Demo application (at least 3 nodes and 2 IP subnets). Use the available scripts in the saft directory of the svn, try to understand what they are doing.
- Understand the implementation of the SAFT protocol (source code and corresponding output).
- 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 how the SAFT building blocks can be used for other transport protocols.
- Determine all areas where the SAFT protocol needs control mechanism.
- Consider how the control mechanisms will be configured and fine tuned (by hand, RTT measurements, external measurements, etc.).
- Think about possible test scenarios for the functional verification and the performance analysis.

2.2.3 Implementation

- Adhere to the Linux coding style guide [6].
- Port the available SAFT code to ANA 0.2.
- Implement the support for multiple applications on top of SAFT.
- Implement an ANA-UDP.
- Implement the control mechanisms for SAFT. Start with the most important one.
- In several iterations optimize your implementation.
- Provide a simple validation script, that determines whether your Bricks work correctly.
- Optional: Try whether your implementation works also in the Linux kernel space.
- Optional: Build an ANA-TCP from the available building blocks.

2.2.4 Validation

- Validate the correct operation of your implementation.
- Check the resilience of the implementation, including its configuration interface, to uneducated users.
- 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

- Document your code with doxygene [8] according to the ANA guidelines.
- Write a documentation about the design, implementation and validation of SAFT in ANA.
- Use the default ANA layout for all your figures.

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] The Transport Layer Revisited, Simon Heimlicher, Rainer Baumann, Martin May, Bernhard Plattner, IEEE COMSWARE 2007, Bangalore, India, 2007
[5] Diego Adolf: Implementation of SAFT on ANA [TIK SA-2008-12]
[6] Available on your Linux box: file:///usr/src/linux/Documentation/CodingStyle
[7] https://www.ana-project.org/wiki
[8] http://www.stack.nl/~dimitri/doxygen/

References

[1] J. Postel, "Transmission Control Protocol." RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[2] J. Postel, "Internet Protocol." RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[3] "Autonomic Network Architecture (ANA)." Source code available: `http://www.ana-project.org` (URL last accessed on 2009-01-18).

[4] Z. Fu, X. Meng, and S. Lu, "How Bad TCP Can Perform in Mobile Ad-hoc Networks," *Computers and Communications, 2002. Proceedings. ISCC 2002. Seventh International Symposium on*, pp. 298–303, 2002.

[5] S. Heimlicher, R. Baumann, M. May, and B. Plattner, "SaFT: Reliable Transport in Mobile Networks," *Mobile Adhoc and Sensor Systems (MASS), 2006 IEEE International Conference on*, pp. 477–480, Oct. 2006.

[6] S. Heimlicher, R. Baumann, M. May, and B. Plattner, "The Transport Layer Revisited," *Communication Systems Software and Middleware, 2007. COMSWARE 2007. 2nd International Conference on*, pp. 1–8, Jan. 2007.

[7] S. Heimlicher, P. Nuggehalli, and M. May, "End-to-end vs. Hop-by-hop Transport," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 3, pp. 59–60, 2007.

[8] D. Adolf, "Implementation of SAFT on ANA." semester thesis, ETH Zurich, 2008.

[9] G. Bouabene, C. Jelger, and A. Keller, "ANA Core Documentation D1.10."

[10] "ANA Blueprint - First Version Updated." `http://www.ana-project.org/deliverables/2007/ana-d1.456-final.pdf` (URL last accessed on 2009-01-29).

[11] R. Gold, P. Gunningberg, and C. Tschudin, "A virtualized link layer with support for indirection," in *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, (New York, NY, USA), pp. 28–34, ACM, 2004.

[12] S. Heimlicher, "SaFT Store and Forward Transport. Reliable Transport in Wireless Mobile Ad-hoc Networks," Master's thesis, ETH Zurich, 2005.

[13] J. D. Day and H. Zimmermann, "The osi reference model," *Proceedings of the IEEE*, vol. 71, pp. 1334–1340, Dec. 1983.

[14] V. Jacobson, "Congestion Avoidance and Control," in *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, (New York, NY, USA), pp. 314–329, ACM, 1988.

[15] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach, 3rd Edition (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann, 3 ed., May 2003.

[16] "Virtual Box." Source code available: `http://www.virtualbox.org` (URL last accessed on 2009-01-18).

[17] "VMWare." Software available: `http://www.vmware.com` (URL last accessed on 2009-01-18).

[18] "Ubuntu Linux." Source available: `http://www.ubuntu.com` (URL last accessed on 2009-01-18).