KTH Computer Science
and Communication

Provoking software failures in SIP servers – the development of an automatic test tool

MIKAEL ELIASSON

Master's Thesis at NADA
Supervisor: Karl Meinke
Examiner: Stefan Arnborg

TRITA xxx yyyy-nn

Abstract

The Session Initiation Protocol (SIP) is a signaling protocol for Internet telephony, instant messaging and alike, based on a request/response transaction model. SIP elements capable of receiving requests and taking appropriate action are known as SIP servers. The SIP standard allows a great deal of freedom in the construction of a valid SIP message. This freedom implies that SIP servers are likely to receive a great deal of unexpected SIP input.

This thesis describes the development of a tool supposed to test the robustness of various SIP server implementations. The aim of the tool is to provoke software failures in the servers it tests and determine the reasons for the failures. This thesis focuses on the tool's ability to provoke failures. The tool attempts to provoke failures in a server by feeding it with unexpected SIP input. The unexpected input is generated by configuring other SIP elements to send SIP messages intended for the server under test (SUT) through the tool. Before the tool forwards a received message to the SUT, it modifies the message in order to make it unexpected. The modifications that the tool does to messages are random, governed by a set of rules and probabilities that are defined and set prior to execution. The tool treats the SUT as a black box, sending messages to it using the same interfaces as any other SIP element would use. In order to detect failures, the tool monitors the SUT. As soon as the test tool succeeds in provoking a failure, it attempts to determine the reason for the failure. The test tool is developed in C++ using a modular design approach, allowing new functionality to be added easily. The result is a fully automated test tool, capable of provoking software failures in arbitrary SIP servers.

Referat

Provocering av mjukvarufel i SIP-servrar – utvecklingen av ett automatiskt testverktyg

SIP (Session Initiation Protocol) är ett protokoll som används för att upprätta sessioner på Internet mellan två eller flera användare. SIP bygger på ett förfrågan/svar-förfarande där SIP-element med förmågan att ta emot och hantera SIP-meddelanden som är förfrågningar kallas SIP-servrar. SIP-standarden tillåter att korrekta SIP-meddelanden kan formuleras på en rad olika sätt, vilket resulterar i att SIP-servrar med hög sannolikhet får ta emot oväntade SIP-meddelanden.

Denna rapport beskriver utvecklingen av ett verktyg avsett att testa SIP-servrar med avseende på robusthet. Verktyget har som mål att provocera mjukvarufel i de servrar det testar och, ifall det lyckas provocera ett fel, avgöra vad som orsakade felet. Den här rapporten fokuserar på verktygets förmåga att provocera fel. Verktyget försöker provocera fel i en server genom att skicka oväntade SIP-meddelanden till den. Oväntade SIP-meddelanden genereras genom att konfigurera andra SIP-element att skicka SIP-meddelanden till servern under test (SUT) genom verktyget. Innan verktyget skickar ett meddelande vidare till SUT så modifierar verktyget meddelandet så att det blir oväntat. Modifieringarna som verktyget gör är slumpmässiga, styrda av en mängd regler och sannolikheter. De olika reglerna och sannolikheterna definieras och sätts av användaren innan verktyget exekveras. När verktyget skickar oväntade SIP-meddelanden till SUT så går det till väga på samma sätt som andra SIP-element som kommunicerar med SUT gör. Verktyget övervakar SUT för att avgöra ifall det lyckas med att provocera mjukvarufel. Ifall verktyget upptäcker att det lyckats med att provocera ett mjukvarufel så försöker det genast att avgöra vad som orsakade felet. Verktyget är implementerat i C++. Den modulära arkitekturen gör det möjligt att enkelt addera ny funktionalitet till verktyget. Resultatet är ett helt automatiskt verktyg, kapabelt att provocera mjukvarufel i godtyckliga SIP-servrar.

Contents

Chapter 1

Introduction

In this chapter the Master's project is introduced. The problem that was investigated and the background to it are described in the first three sections. The fourth section gives an outline of the rest of this Master's thesis.

1.1 Background

The Session Initiation Protocol (SIP) is an application-layer control protocol for creating, modifying, and terminating sessions with one or more participants. Examples of sessions are Internet telephone calls, multimedia distribution, and multimedia conferences. SIP is based on a request/response transaction model in which SIP clients and SIP servers interact. Each transaction consists of a request, sent by a SIP client to a SIP server. The request invokes a method, or function, on the SIP server and at least one response.

A variety of SIP servers can, and often do, exist in a SIP architecture. Ericsson AB developes a proxy server known as the Call/Session Control Function (CSCF). Proxy servers help route requests to the user's current location, authenticate and authorize users for services, implement provider call-routing policies, and provide features to users. The CSCF is a heavy-weight proxy server, often considered to be the heart of the IP multimedia subsystem (IMS – an architectural framework for delivering internet protocol (IP) multimedia to mobile users).

The SIP standard allows a great deal of freedom in the construction of a valid SIP message. This freedom implies that SIP servers in general are likely to receive a great deal of unexpected SIP input. Ericsson AB therfore wanted to have an automatic test tool, that attempts to provoke software failures in the CSCF, developed. It was a requirement that the test tool should attempt to provoke software failures in the CSCF by feeding it with unexpected SIP input. An additional requirement, in case the test tool succeeds in provoking a software failure, was that the test tool should have the ability to determine the reason for the failure. It was also a wish that the test tool, in order to enable testing of arbitrary SIP servers, was designed to be as generic as possible.

The test tool that was developed can be seen as having two different modes. One mode in which it provokes software failures, and one mode in which it determines the reason for a provoked software failure. The development of the test tool was split into two Master's projects. This Master's project focused on the mode in which the test tool generates unexpected SIP input and feeds it to a SIP server in order to provoke software failures. The mode in which the test tool determines the reason for a provoked failure was the focus of the other Master's project.

1.2 Problem

In order to develope an automated test tool with the ability to provoke software failures in arbitrary SIP servers, a number of questions need to be answered. The first question, which must be kept in mind when answering the rest of the questions, concerns the actual design of the test tool:

> *How can the architecture of a test tool, generic enough to test arbitrary SIP servers, be defined?*

The test tool is required to provoke software failures in the subject under test by injecting it with unexpected SIP. The second question concerns the nature of the unexpected SIP input:

> *What is unexpected SIP input?*

The third question concerns how the test tool should get hold of unexpected SIP input:

> *How can the unexpected SIP input be generated?*

The fourth question concerns how the test tool should connect to the SUT in order to inject unexpected SIP input:

> *How should the test tool inject the unexpected input into the SIP server under test?*

1.3 Aim

The aim of this Master's project was to develop a tool, capable of testing the robustness in arbitrary SIP servers by injecting unexpected SIP input into them. The tool was meant to be an effective complement to manual testing of SIP servers. As a complement to manual testing, the test tool had to be automated and easy to use.

1.4 Thesis outline

The rest of this thesis is outlined as follows: Chapter 2 gives an overview of the environment in which the test tool operates. In chapter 3 the different methods used in this Master's project are described and the usage of them are motivated. Chapter 4 exemplifies earlier work related to testing of SIP servers. The methods used by the test tool to generate and send unexpected SIP input are described in 5. The developed tool is presented in chapter 6 and evaluated in chapter 7. Finally, in chapter 8 conclusions concerning the test tool and the development of it are drawn.

Chapter 2

Session Initiation Protocol (SIP)

This chapter gives an overview of the Session Initiation Protocol (SIP). The first four sections are basically a very short summary of Rosenberg et al. (2002), describing SIP in general and focusing on areas that affects the design of the test tool. The fifth section presents an architecture that uses SIP.

2.1 Overview

Many applications of the Internet require the creation and management of a session, where a session is considered an exchange of data between an association of participants. SIP enables Internet endpoints (called user agents) to discover one another and to agree on the character of a session they would like to share. For locating potential participants, and for other functions, SIP enables the creation of an infrastructure of network hosts (called proxy servers) to which user agents can send registrations, invitations to sessions, and other requests. SIP is a tool for creating, modifying, and terminating sessions that works independently of both underlying transport protocols and the type of session that is being handled. SIP is considered to be the protocol that will merge together the cellular and the Internet worlds [Camarillo, 2002].

2.2 Functionality provided by SIP

SIP is an application layer protocol for establishing, modifiying and terminating sessions. SIP is independent of the type of session being handled. In short, SIP is used to distribute session descriptions among potential participants. Once the session description is distributed, SIP can be used to negotiate and modify the session parameters and terminate the session. A session parameter can for example be the type of codec used to encode voice in an audio-video session.

SIP supports user mobility. A user in a SIP environment is identified by a SIP uniform resource identifier (URI). The format of a SIP URI is similar to an email address and can look like the following:

```
sip:alice@wonderland.com
```

SIP is designed to be used together with other protocols. Examples of other protocols are the Real-time Transport Protocol (RTP) for transporting real-time data, and the Session Description Protocol (SDP) for describing multimedia sessions.

SIP does not provide services. Rather, SIP provides primitives that can be used to implement different services. For example, SIP can locate a user and deliver an opaque object to his current location. If this primitive is used to deliver a session description written in SDP, the endpoints can agree on the parameters of a session.

2.3 Structure of the protocol

SIP is request/response protocol, based on the Hypertext Transfer Protocol (HTTP). A client is an element that generates SIP requests. A server is an element that receives SIP requests and returns SIP responses.

SIP is structured as a layered protocol. Not every element specified by the protocol contains every layer. Furthermore, the elements specified by SIP are logical elements, not physical ones. A physical realization can choose to act as different logical elements. The transport layer is a highly relevant layer in the context of this thesis. It defines how a client sends requests and receives responses and how a server receives requests and sends responses over the network. All SIP elements contain a transport layer, and the developed test tool more or less also contains one.

2.4 SIP messages

SIP is a text-based protocol. A SIP message is either a request from a client to a server, or a response from a server to a client. Both types of messages consist of a start line, one or more header fields, an empty line indicating the end of the header fields, and an optional message body. The start line, each message header line, and the empty line must be terminated by a carriage-return line-feed sequence (CRLF). The empty line must be present even if the message body is not.

2.4.1 Requests

SIP requests are distinguished by having a request line as start line. A request line contains a method name, a request URI, and the protocol version separated by a single space character. The method indicates the type of request, the request URI indicates where the request has to be routed, and the protocol version indicates which version of SIP that is being used. Seven fundamental methods are listed in table 2.1. All of the methods, except MESSAGE which is defined in Campbell et al. (2002), are defined in Rosenberg et al. (2002). A SIP request line can look like the following:

```
INVITE sip:alice@wonderland.com SIP/2.0
```

Method	Description
REGISTER	For registering contact information.
INVITE	For setting upp sessions.
ACK	For setting upp sessions.
CANCEL	For setting upp sessions.
BYE	For terminating sessions.
OPTIONS	For querying servers about their capabilities.
MESSAGE	For instant messaging between SIP elements.

Table 2.1. Each SIP request invokes a particular method. The methods listed here are seven of the most basic ones.

Status code	Response class
100 – 199	Provisional – request received, continuing to process the request.
200 – 299	Success – the action was successfully received, understood, and accepted.
300 – 399	Redirection – further action needs to be taken in order to complete the request.
400 – 499	Client error – the request contains bad syntax or cannot be fulfilled at this server.
500 – 599	Server error – the server failed to fulfill an apparently valid request.
600 – 699	Global failure – the request cannot be fulfilled at any server.

Table 2.2. The different SIP message response classes.

2.4.2 Responses

SIP responses are distinguished by having a status line as start line. A status line contains the protocol version, a numeric status code and its associated reason phrase separated by a single space character. The status codes indicates the outcome of an attempt to understand and satisfy a request. They status codes are integers that range from 100 to 699 and are grouped into classes. The different classes can be seen in table 2.2. The reason phrase is intended to give a short textual description of the status code, intended for the human user. A SIP response line can look like the following:

```
SIP/2.0 180 Ringing
```

Note that many of the header fields in a SIP request are copied from the related SIP request.

2.4.3 Header fields

Each header field consists of a field name followed by a colon and the field value. An arbitrary amount of whitespace is allowed on either side of the colon. Header fields can be extended over multiple lines by preceding each extra line with at least one single space character or horizontal tab. The format of a header field-value is defined per header-name. It will always be a combination of whitespace, tokens, separators, and quoted strings. Some header fields only make sense in requests or responses. These are called request header fields and response header fields, respectively. If a header field appears in a message not matching its category, for example a request header field in a response, it must be ignored. SIP allows header field names to be represented in an abbreviated form.

2.4.4 Bodies

The body carried by a message is usually a session description, but it can consist of any opaque object. Requests may contain message bodies. The interpretation of the body depends on the request method. For response messages, the request method and the response status code determine the type and interpretation of any message body. All responses may include a body. Messages can carry several bodies.

2.4.5 Examples of SIP messages

In this section two complete SIP messages can be seen. The first SIP message is a typical request invoking the MESSAGE method:

```
MESSAGE sip:user_a@130.100.96.240 SIP/2.0
Via: SIP/2.0/UDP 130.100.96.113:5060;branch=z9hG4bK42.12ae.1
To: "User A" <sip:user_a@130.100.96.240>
From: "User B" <sip:user_b@130.100.96.113>;tag=14c5a5c-356e.1
Call-ID: 71e5-14a34-5f631@130.100.96.113
CSeq: 24693 MESSAGE
Max-Forwards: 70
Content-Length: 13
Content-Type: text/plain
Expires: 0

Hello, World!
```

The instant message that is delivered with this SIP message is “Hello, World!”. It is carried in the body of the SIP message. The second SIP message, which is a 200 OK, is a typical response to the SIP request above:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 130.100.96.113:5060;branch=z9hG4bK42.12ae.1
To: "User A" <sip:user_a@130.100.96.240>;tag=24962cc-7fc9.6
```

```
From: "User B" <sip:user_b@130.100.96.113>;tag=14c5a5c-356e.1
Call-ID: 71e5-14a34-5f631@130.100.96.113
CSeq: 24693 MESSAGE
Content-Length: 0
```

By comparing the two SIP messages, we can see that several header fields are copied from the SIP request to the SIP response. In this example it is the Via, To, From, Call-ID, and CSeq header fields that are copied.

2.5 SIP elements

Several different SIP elements can exist inside a SIP environment. This section describes the fundamental ones.

2.5.1 User agent

A user agent (UA) represents an endpoint in a SIP environment. It can act as both a user agent client (UAC) and user agent server (UAS). A UAC is capable of generating a request and processing a response. A UAS is capable of receiving request, servicing them, and generating responses.

2.5.2 Registrar

A registrar is a server that accepts REGISTER requests and stores the information it receives in those requests. The information can then be used by a SIP redirect or proxy server to obtain information about a callee's possible locations.

2.5.3 Redirect server

A redirect server is a UAS that helps locate other user agents by providing alternative locations where the user can be found. A redirect server does not take any action to locate a user, it only returns a list of possible locations where the user might be.

2.5.4 Proxy server

A proxy server is an intermediary entity that acts as both a server and a client for the purpose of making requests on behalf of other clients. A proxy server primarily plays the role of routing, which means its job is to ensure that a request is sent to another SIP element "closer" to the targeted user. Proxies are also useful for enforcing policy (for example, making sure a user is allowed to make a call). A proxy interprets, and, if necessary, rewrites specific parts of a request message before forwarding it.

2.6 Sending and receiving SIP messages

The SIP transport layer, mentioned in section 2.3, is responsible for the actual transmission of requests and responses over network transports. The client side of the transport layer is responsible for sending the requests and receiving the responses. Before a request is sent, the client transport must insert a value of the "sent-by" field into the Via header field. This field contains an IP address or host name, and port. The transport layer must be prepared to receive an incoming connection on the source IP address from which the request was sent and port number in the "sent-by" field.

For any port and interface that a server listens on for UDP, it must listen on that same port and interface for TCP. This is because a message may need to be sent using TCP, rather than UDP, if it is too large.

When the server receives a request, it must examine the value of the "sent-by" parameter in the top Via header field value. If the host portion of the "sent-by" parameter contains a domain name or an IP address that differs from the packet source address, the server must add a "received" parameter to that Via header field value. This parameter must contain the source address from which the packet was received. The server transport uses the value of the top Via header field in order to determine where to send a response. If the top Via header field has a "received" parameter, the response must be sent to the address in the "received" parameter, using the port indicated in the "sent-by"value. Otherwise, the response must be sent to the address indicated by the "sent-by" value.

2.7 An architecture that uses SIP

The IP Multimedia Subsystem (IMS) is an architectural framework for delivering internet protocol (IP) multimedia to mobile users. The aim of IMS is to a merge the Internet with the cellular world, making the mobile Internet paradigm come true [Camarillo and García-Martín, 2006].

The user can connect to an IMS network in various ways. Direct IMS terminals (such as mobile phones and computers) can register directly on an IMS network, even when they are roaming in another network. One requirement is that they can run SIP user agents. Other phone systems, like the old analogue telephones gateways.

The CSCF, which is a SIP server, is an essential node in IMS. It is the first point of contact for the IMS terminals. It sits on the path of all signaling messages, and can inspect every message. Another important node is the application server (AS). It host and executes services, and it communicates with the CSCF using SIP.

Chapter 3

Method survey

This chapter describes, and motivates the usage of, the methods that were applied in this Master's project. The first section describes methods related to the purpose of the test tool. The second section describes methods related to the actual development of the test tool.

3.1 Methods for software testing

Since the aim of this Master's project was to develop a tool with the purpose of testing SIP servers, different methods for software testing were studied. Below, a description of each method that was deemed as appropriate for the test tool to use, together with an explanation why it was deemed as appropriate, is given.

3.1.1 Testing for error handling and recovery

It was a requirement from Ericsson AB that the test tool should test a SIP server by injecting unexpected SIP input into it. The purpose of this approach is to test the subject under test (SUT) with respect to error handling and recovery. It is often used as a complement to testing for correctness of functionality. Testing for correctness of functionality can be seen as examining a program to see if it does not do what it is supposed to do whereas testing for error handling and recovery can be seen as examining whether the program does what it is not supposed to do.

Obviously the number of things that a program should do is finite and the number of things that it should not do is infinite. Therfore, the number of tests that examine error handling and recovery typically outnumbers the number of tests that examines correctness of functionality. In mature test suites, tests that examine error handling and recovery outnumber tests that examines correctness in a ratio of 4:1 or 5:1 [Beizer, 1995].

Note also that tests representing unexpected and invalid input conditions, which is just the kind of tests that the test tool should produce, seem to have a higher error-detection yield than do tests for valid input conditions [Myers et al., 2004]. It can be concluded that the set of tests that the test tool should be able to produce is as important as it is large.

3.1.2 Black box testing

Black box testing is testing without knowledge about the internal structure of the subject under test (SUT). It can be seen as testing with respect to the specification of the SUT, interested only in what the answers are, not in how the SUT arrives at them.

Since the test tool attempts to provoke software failures in SIP servers by feeding them with unexpected SIP input, it basically needs to know only two things: how to produce unexpected SIP input and how to feed the input to a SIP server. How to produce the unexpected input can be deduced by looking at the definition of the SIP protocol. The SIP protocol defines what expected input is and everything that is not expected is unexpected. How to feed the unexpected input to a SIP server can be deduced by looking only at the external structure of the SIP server, which should conform to the definition of the transport layer in the SIP protocol.

It can be concluded that the test tool can treat the SUT as a black box since no knowledge about the internal structure of the SIP server is needed. It can also be concluded that the SUT should, if possible, be treated as a black box. One strong reason for this is that it keeps the test tool generic. As long as a SIP server conforms to the definition of the transport layer in the SIP protocol it can be tested by the tool. Another reason is that a SIP server implementation can be extremely complex and an automatic test tool with knowledge about the internal structure of such a complex SIP server will probably also have to be very complex.

The opposite to black box testing approach is an approach known as white box testing. It uses tests based on the internal structure of the SUT and was therefore never a method that the test tool could have used since it should be able to test arbitrary SIP servers.

3.1.3 Random testing

The input domain of almost any nontrivial software application is infinite and complete testing is impossible [Myers et al., 2004]. Not only are there lots of individual inputs, but inputs can be combined in so many different combinations that it is impossible to apply them all. However, all elements are seldom equally interesting to apply and one way to shrink the size of the input domain is to apply the right set of inputs. The right set of inputs can for example be the set of inputs which is most likely to occure in reality. But still, if the right set of inputs also is large, the problem with an infinite input domain remains.

When the input domain is so large that it is not feasible to manually design tests to cover even a small part of it, an alternative approach is random testing. What random testing does is simply to randomize the input. The major advantage with the random testing method is that it is straightforward to automate which results in fast generation of tests. One possible disadvantage is that the randomized input might not be uniformly distributed which leads to the fact that large parts of the input domain might be missed in the generation of test cases [Mankefors et al., 2003].

The input domain of a SIP server is vast. Unexpected input, which is what the test tool uses when testing a SIP server, can be just about anything. Therefore, manually designing tests to attempt to cover as much of the input domain as possible is not an effective approach. Manually designed tests can instead be very useful when covering very specific subsets of a large input domain.

The random testing method is, on the other hand, a very feasible approach when attempting to cover as much as possible of a SIP server's input domain. SIP messages, which are the smallest elements in the unexpected input a SUT will receive, are easy to do random modifications on. Tasks involving randomness, like randomly deciding which parts of a SIP message to delete or replace, are easily accomplished since a SIP message can be regarded as an array of characters. The tasks are also easily automated, allowing fast creation of test cases. Furthermore, the tasks can easily be steered to cover different subsets of the input domain to avoid an uneven distribution of the tests.

It can be concluded that the random testing method is an efficient approach when attempting to cover as much as possible of the input domain of a SIP server if the different tests are automatically generated. Note that in this Master's project only a small fraction of the input domain is likely to cause software failures in the SUT. Since the tests that the test tool produces are randomized, the test tool is expected to have to produce a large number of tests in order to provoke software failures in the SUT.

3.1.4 Test automation

In general, software testing requires that a large number of tests are generated, executed and afterward inspected for correctness. The large number of tests makes automated testing an attractive alternative to manual testing, since manual testing is labor intensive, expensive and error prone [Bieman and Yin, 1992]. Test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions

It has already been concluded in section 3.1.3 that the number of possible tests, when testing how well a SIP server handles unexpected input, will always be extremely large. Obviously, automated generation, execution and inspection of the possible tests is therefore a desirable feature of the tool. It can be concluded that the tool should be as automated as possible.

3.2 Software development methods

Different software development methods were studied in order to find one that was appropriate to use to develope the test tool. The methods that were judged as

Principle	Description
Small releases	The minimal useful set of functionality is developed first. New versions with added functionality are released often.
Simple design	Design is carried out to meet the current requirements and no more.
Test-first development	A unit test framework is used to write tests for a new piece of functionality before the functionality is implemented.
Refactoring	Developers are expected to refactor as soon as possible code improvements are found.
On-site customer	The customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Table 3.1. The main principles of extreme programming.

appropriate are described in this section. Along with each method description is the reasoning which led to the conclusion that the method was appropriate.

3.2.1 Extreme programming

Extreme Programming (XP) is an agile software development method. As such, XP allows the development team to focus on the software itself rather than on its design and documentation. XP relies on an iterative approach to software development. The method is designed to support rapid changes of system requirements during the development process, and to deliver working software quickly to the customer. Some of the most important XP principles can be seen in table 3.1.

The development of the test tool was, as mentioned in section 1.1, split into two Master's projects which were conducted in parallel. The team that developed the test tool therefore consisted of three members: myself (Mikael Eliasson), the student responsible for the other Master's project (Carl Lundin) and our employer at Ericsson AB (Ph.D. Johnny Bigert).

It was at an early stage of the development decided that a simple test tool working in its proper environment was preferable to a test tool with lots of different features. It was also decided that all produced code should be tested, and that test-first development therefore would be a sound approach.

These decisions, and the fact that the role of our employer in the development team was that of an on-site customer, coincides well with the XP principles listed in table 3.1. Therefore, the development team came to the conclusion that XP was an appropriate software development method to use.

An example of a method that was discarded is the waterfall model. It was discarded mainly because it is a sequential software development method. The major problem with sequential methods is the inflexible partitioning of the project into distinct stages [Sommerville, 2004]. This inflexibility makes it difficult to respond to changing system requirements.

3.2.2 Object-oriented design

Object-oriented design is design structured as interacting objects. Objects can be seen as independent machines with their own responsibilities. Each object has the ability to receive, process, and send data to other objects.

By applying object-oriented design in the development of the test tool, it was made sure that the test tool will be easier to adapt to changing, and new requirements. New objects, bringing new functionality to the tool, can easily be added to the set of objects that constitutes the tool, while old objects can be replaced with new, better ones.

3.2.3 C++

The test tool was implemented using C++. The usage of C++ was a requirement from Ericsson AB.

Chapter 4

Earlier work

This chapter gives an overview of/exemplifies earlier work in areas related to the area of this Master's thesis.

4.1 PROTOS Test-Suite

PROTOS test-suite uses the black box method for testing the robustness of SIP server implementations. Test cases are injected through interfaces and the SIP element under test is monitored for failures. The test-suite tests SIP over UDP and consists of 4527 test cases. Each test case consists of SIP INVITE message containing one or several exceptional elements, where an exceptional element is a piece of data designed to provoke undesired behavior in the SIP element under test. The design of the exceptional elements are based on experience gathered from previous test suites and publicly reported vulnerabilities [Wieser et al., 2004].

The test cases are arranged into 54 different test groups, each test group focusing on a certain part of a SIP INVITE message. The test cases are injected into the SIP element under test by a UDP injector. In order to enable test automation, connection teardown to cancel previous INVITE requests is implemented. The connection teardown is accomplished by injecting a CANCEL and an ACK after each test case. If connection teardown had not been implemented, test execution against terminals would have required manual intervention to terminate incoming calls from each test case.

By sending a valid SIP INVITE message between every test case, the SIP element under test is monitored for failure. If no response to a valid INIVTE is detected from the SIP element under test, it has failed.

4.2 SipBomber

SipBomber is a tool for testing SIP server implementations. It supports testing over both UDP and TCP. The tool supports automatic and manual testing. The user of the tool has the ability to create and run custom tests. SipBomber analyzes responses from the SIP server under test for compliance with Rosenberg et al. (2002). SipBomber has a graphical user interface and is developed for the LINUX platform.

Chapter 5

Provoking failures in SIP servers

This chapter describes the method used by test tool when it attempts to provoke software failures in SIP servers.

5.1 Generating unexpected SIP input

Exactly what unexpected SIP input constitutes of is determined by the subject under test (SUT), since input that is unexpected by one SIP server might be expected by another. As a consequnce the test tool, which is designed to test different SIP server implementations, must be flexible in its generation of unexpected SIP input.

Let A be the set of all possible SIP input. A can be partitioned into the two subsets B and C, where B is the set of expected SIP input and C is the set of unexpected SIP input. The test tool then fulfills the required flexibilty by being able to create more or less every member of A and letting the user determine how A is to be partitioned into B and C.

Let D be the set of unexpected SIP input similar to expected SIP input. D is more often than not one of the more important subsets and therefore likely to be the right set of inputs (see section 3.1.3 for a discussion concerning the right set of inputs). One reason for this is that members of D are likely to occure in reality, for examlpe due to flawed SIP element implementations. Another reason is that members of D often can be difficult to distinguish from expected SIP input, and are therefore treated as expected input by SIP servers. This mistreatment can in turn lead to errouneos and unpredictable behavior in the SIP servers.

Since D is considered such an important subset of C to test, the test tool supports the generation of members of D. It accomplishes this by always using members of B as starting point when generating members of C. Members of B can, by being modified in simple ways, easily be transformed into members of D. Figure 5.1 shows the sets A,B, C and D related to each other in a Venn diagram.

The tool does not create members of B from scratch itself. Instead, the test tool lets other SIP elements that communicates with the SUT send their output meant for the SUT through the test tool. As a result, the test tool has unlimited

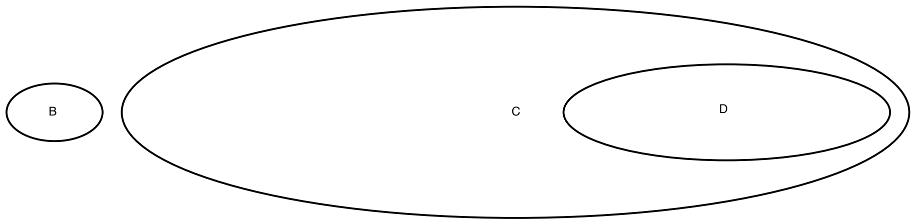


Figure 5.1. B is the set of expected SIP input, C is the set of unexpected SIP input, and D is the set of unexpected SIP input that is similar to expected input. Note that $A = B \cup C$ where A is the set of all possible SIP input.

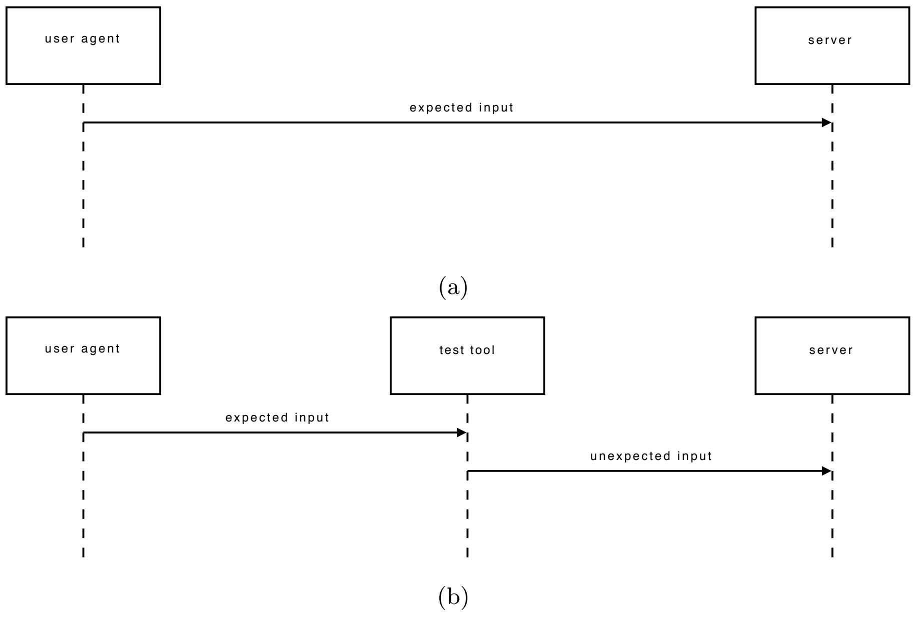


Figure 5.2. Figure 5.2(a) shows a basic scenario in a SIP architecture: a SIP user agent sending input to a SIP server. Figure 5.2(b) shows a possible placement of the test tool in that scenario.

access to elements in B. Figure 5.2 shows one such possible scenario. Figure 5.2(a) shows the scenario prior to execution of the test tool, figure 5.2(b) shows the same scenario when the test tool running.

In the scenario in figure 5.2(b) the user agent is configured to treat the test tool as a SIP proxy server. It follows that the user agent will send members of B to the test tool. A reason for this approach is that it makes it possible to keep the tool simple. To be able to create elements of B from scratch, the tool would have to implement a fully featured SIP user agent. Functionality of that kind would bring unnecessary complexity to the test tool and be a likely source for errouneous behaviour. The modifications that the test tool does to the expected SIP input that is sent through it are automated, random modifications. In chapter 3 the decision to use automated, random modifications to generate unexpected SIP input is motivated in detail. In short, it is an efficient way to cover as much as possible of a large set of inputs, in this case to cover as much as possible of C or certain subsets of C.

The modifications that the test tool does to elements of B are not completely random, they are governed by rules and probabilities. The rules are used to focus on different subsets of C. The probabilities are used to determine how often a certain rule is enforced. This makes it possible to concentrate on very specific subsets of C. The user determines which rules that the test tool will use, and to what extent each rule will be applied. It is possible to add new rules to the existing set of rules.

Note that the probability to provoke a software failure in a SIP server, by sending a SIP message containing random modifications, is in general low. Therefore, the test tool should be prepared to send a large number of SIP messages to the SUT in order to provoke a software failure. It is possible to let several SIP elements, communicating with the SUT, send their SIP output addressed to the server through the test tool. This feature sees to that the test tool should have unlimited access to elements of B at all times and therefore never have to sit idle.

The test tool also has the ability to determine from which SIP element that a received element of B was sent. It can therefore treat elements of B sent by one SIP element different from those sent by another, applying different rules and probabilities to the members of B depending on where they come from.

As a result, the test tool can let different SIP elements focus on different subsets of C. If, for example, two SIP elements are configured to send their SIP output addressed to the server through the test tool, one SIP element can be used to cover as much of D as possible while the other SIP element is used to cover another subset of input that is desirable to test.

5.2 Feeding unexpected SIP input to SIP servers

In order to feed unexpected input to the subject under test (SUT), the test tool uses the same public interface as any other element “talking SIP” would use to communicate with a SIP server. By feeding the SUT with unexpected input through its public SIP interface, the test tool does not need any knowledge about the internal workings of the SUT and can consequently treat the SUT as a black box. In addition, the server is not aware of that it is being tested since the test tool does not act any different than other SIP elements communicating with the server.

As mentioned in section 2.6, a SIP server that receives a SIP request will examine the value of the “sent-by” parameter in the top Via header field value. If it differs from the source address of the packet that contained the request, the SIP server will send consequent responses to the packet source address. As a result, the SIP server under test will send SIP responses to the test tool. How the scenario that was depicted in figure 5.2(b) in section 5.1 looks like when a response sent by the SUT is included can be seen in figure 5.3

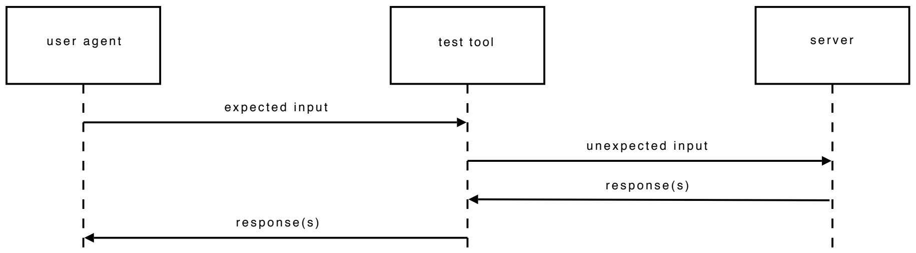


Figure 5.3. The test tool sends unexpected input to the SUT. The SUT, according to the routing mechanisms of SIP, sends responses back to the test tool.

It is then the test tool's responsibility to route the responses to the correct SIP elements. If the test tool would have to examine the content of SIP messages, it would have to have a lot of functionality in common with regular SIP servers. This would obviously not be a good feature of the test tool which should be guaranteed to be stable where the SUT might not be. Instead, the test tool accomplishes the routing of received SIP messages solely by examining the source and destination addresses of the IP packets containing the SIP messages. Note that by using this routing approach, the test tool puts one constraint on the SIP environment it operates in: it must be possible to tell where a SIP message should be sent just by looking at the source and/or destination address of the packet that contained the SIP message.

The fact that the SUT will send responses to the test tool has a positive consequence: the test tool can determine, by looking at the responses received from the SUT, how well the SUT handles the unexpected input that is sent to it. The test tool can for example use the respones to detect software failures in the same way that the Protos test-suite, that was described in section 4.1, does.

Chapter 6

Design of the test tool

This chapter describes the design of the test tool. As mentioned in section 1.1, the development of the test tool was split into two Master's projects. The parts of the test tool that was developed in this Master's project are described in detail. For detailed descriptions of the parts that were not developed in this Master's project, see Lundin (2007).

6.1 Test tool overview

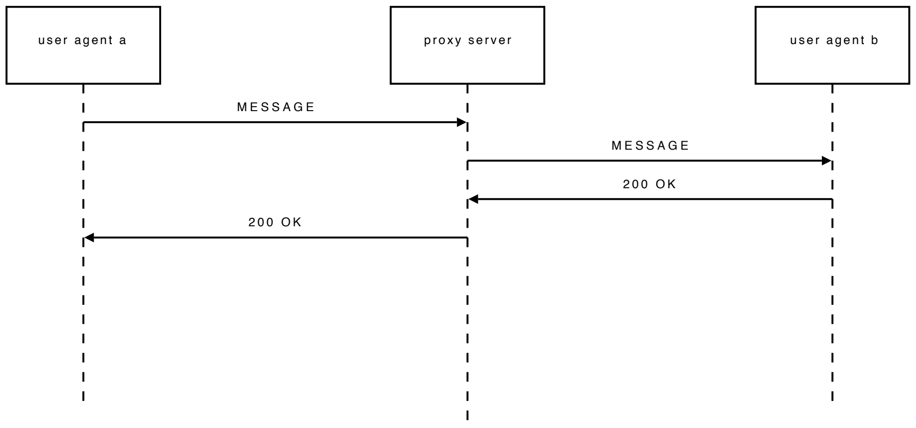
The test tool is a multithreaded [Tannenbaum, 2001] network application. It was developed in C++ using an object-oriented approach. The purpose of the test tool is to provoke software failures in SIP servers and determine the reasons for the failures that it manages to provoke. As soon as the test tool is executed it is fully automated. Prior to execution, the test tool is configured via an XML file.

6.1.1 Placement

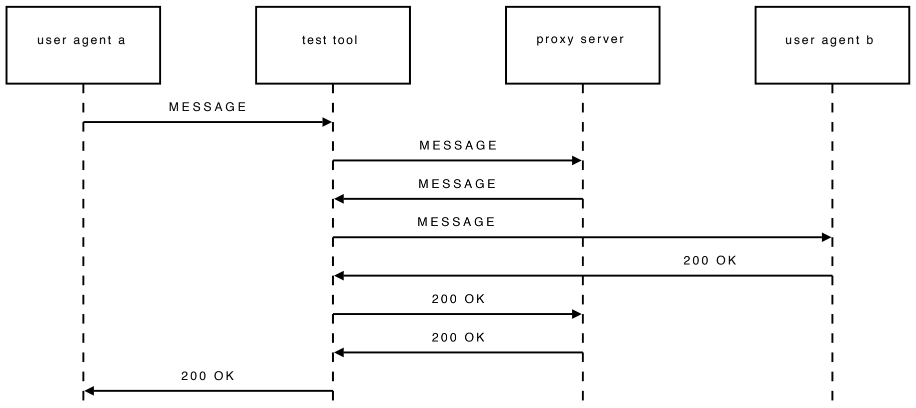
The test tool is designed to co-exist with other SIP elements in an arbitrary SIP architecture. An example of the placement of the test tool in a SIP architecture can be seen in figure 6.1. Two SIP elements, user agent a and user agent b communicates with each other through a SIP proxy server. Figure 6.1(a) depicts the scenario without involving the test tool. In figure 6.1(b) both user agents are configured to send their output meant for the proxy server, which is the subject under test in this scenario, through the test tool.

6.1.2 Routing of SIP messages

As mentioned in section 5.2, the test tool handles the routing of received SIP messages without examining the content of the SIP messages. Instead, the test tool examines the internet protocol (IP) packets containing the SIP messages. Specifically, it stores the source and destination address of each received packet, where



(a)



(b)

Figure 6.1. Figure 6.1(a) shows a basic scenario in a SIP architecture: a SIP user agent sending input to a SIP server. Figure 6.1(b) shows a possible placement of the test tool in the same scenario.

an address consists of IP address and port number. How the test tool makes use of this information depends on the SIP architecture it is operating in.

As also was mentioned in section 5.2, this routing approach puts one constraint on the SIP architecture it operates in: it must be possible to tell where a SIP message should be sent just by looking at the source and/or destination address of the packet that contained the SIP message.

6.1.3 Modular design

The test tool is split into five fundamental modules. The different modules communicates with each other through well-defined interfaces. Each module consists of one or several objects, where an object is an instance of a C++ class. Figure 6.2 depicts the test tool split into its fundamental modules. In short, the controller

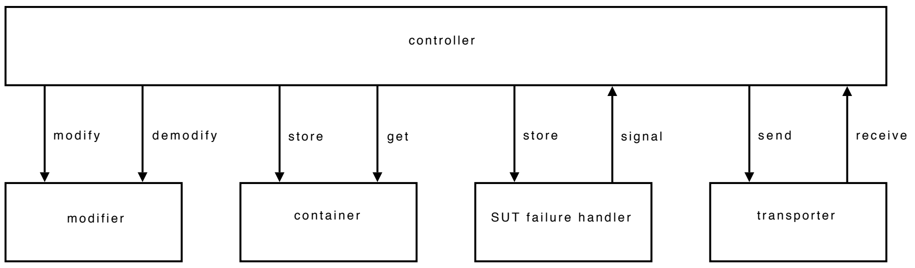


Figure 6.2. The test tool split into its fundamental modules. The arrows indicate the direction of the communication that takes place between the modules.

module is in charge of the other modules and determines the overall behaviour of the test tool. The transporter module receives and sends SIP traffic to and from the test tool. The container module stores SIP traffic that passes through the test tool. The modifier module makes the output which the test tool sends to the subject under test (SUT) unexpected and the SUT failure handler detects when a software failure occures in the SUT. The different modules are described in detail later in this chapter.

The modifier module and the transporter module were developed in this Master's project. The controller module and the container module were developed in Lundin (2007), the other Master's project responsible for the development of the test tool. The development of the SUT crash handler module was interweaved into both of the Master's projects. Each module is described later in this chapter and appendix A contains each modules class diagram. The modules developed in Lundin (2007) will not be described in any detail.

6.1.4 Two different modes

The test tool can be seen as having two different modes. One in which it attempts to provoke software failures in the SUT, and one in which it attempts to determine the reason for a provoked failure. This thesis focuses on the mode in which the test tool attempts to provoke software failures. A typical scenario, when the test tool is operating in that mode, can be seen in figure 6.3. The depicted scenario shows that the test tool as a whole only handles one SIP message at a time.

Another typical scenario when the test tool is in the mode where it attempts to provoke software failures is shown in figure 6.4. The scenario, which takes place in parallel with an arbitrary number of scenarios of the kind described in figure 6.3, depicts the workings of the SUT failure handler module. If the SUT failure handler module detects a relevant software failure it notifies the system controller module of this. The controller module sees to that the tool switches to the mode where it attempts to determine the reason for the software failure.

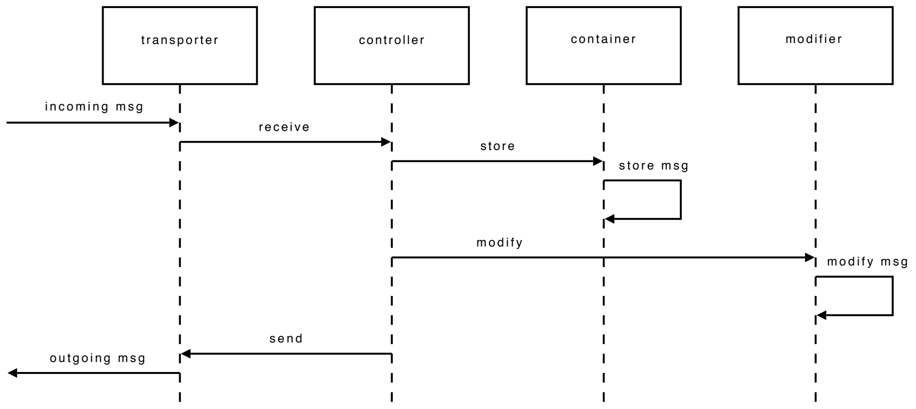


Figure 6.3. A typical scenario when the test tool is attempting to provoke software failures: a SIP message is received, stored, modified, and sent on.

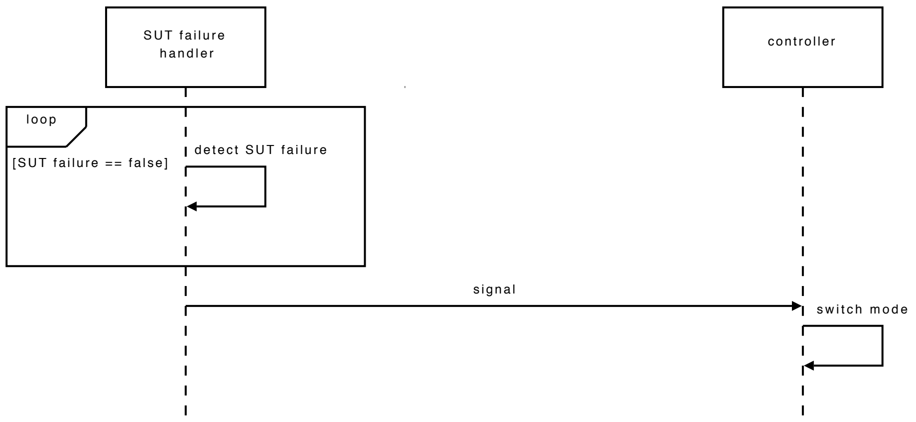


Figure 6.4. A typical scenario when the test tool is attempting to provoke software failures: the test tool has succeeded in provoking a software failure and switches to the mode where it attempts to determine the reason for the failure.

Object	Purpose
modifier(s)	Modify SIP messages in order to make them unexpected, i.e. generate the unexpected input. Know, when modifiying a message, what rules to apply and to which extent each rule is to be applied.
iRuleBase(s)	Define a certain rule that is to be followed when generating unexpected input. Rules are used to focus on different subsets of the set of unexpected input.
sipMsgParser	Assist rules that modifies the content of SIP messages.
demodifier	Undo modifications that the test tool has done to SIP messages, i.e. make the unexpected input expected again.

Table 6.1. The fundamental objects that the modifier module consists of. Every object is an instance of a C++ class.

6.2 Controller module

The controller module manages the overall behaviour of the test tool. Every other module either take their orders from, or reports to, it. The controller module is stateful and operates in different states depending on what it is doing. The only state that is relevant in the context of this thesis, and consequently the only state that will be described, is the state in which the controller module operates when the test tool is attempting to provoke software failures.

A major responsibility that the controller module has in this state is the routing of received SIP messages. It is the controller module that decides, and informs the transporter module of where to send SIP messages. Apart from this external routing, the controller module is also responsible for an internal routing of SIP messages. Each SIP element that sends SIP messages to the test tool is tied to a certain modifier in the modifier module, and it is the controller modules responsibility to which modifier. The controller module was not developed in this Master's project and is therefore not described in any further detail. See Lundin (2007) for additional details. See appendix A for the module's class diagram.

6.3 Modifier module

The modifier module generates the unexpected SIP input that the test tool uses to provoke software failures in the SUT. It accomplishes this by modifying SIP messages that it receives, using different rules and probabilities. The modifier module consists of several objects, which are listed in table 6.1. As indicated in the table, several modifier and iRuleBase objects can exist inside the modifier module.

Several modifier objects can exist since the test tool has, as mentioned in section 5.1, the ability to modify SIP messages received from one SIP element different from those received from another. The test tool accomplishes this by connecting each SIP element that is configured to send SIP messages through the test tool to a certain modifier object. By doing that, the test tool knows which modifier object that is supposed to modify a received SIP message. Each modifier object can then be configured so that no two modifiers treats SIP messages alike.

Several iRuleBase objects can exist since all rules that the modifier module has at its disposal are iRuleBase objects. Actually, all rules are instances of classes that are derived from the IRuleBase class. However, here it suffices to view the rules as iRuleBase objects. The rules that are useful when testing one SIP server may be useless when testing another. The test tool accommodates to these conditions by offering only a small set of predefined rules and instead supports the adding of new, user-defined rules. See appendix A for the module's class diagram.

6.3.1 Modifying SIP messages

The controller module sees to that each received SIP message that is supposed to be modified is sent to the correct modifier object. A modifier object that receives a SIP message uses rules and probabilities to determine how the SIP message should be modified. Each rule defined in the modifier module have three "sub-rules": delete, insert, and replace. It is the "sub-rules" that do the actual modifications to SIP messages. How a modifier object should apply a certain rule is configured by the user with the help of probabilities in four ways: the first probability determines if the rule should be applied at all and the remaining three determines how the "sub-rules" should be applied. An additional probability is used to determine if the modifier should modify the received SIP message at all. A modifier object that receives a SIP message goes through the following steps:

1. Determine if the SIP message should be modified at all. If yes goto 2, else goto 3.
2. For each rule:
    a) determine if the rule should be applied. If yes go to a, else go to b
    b) determine if delete should be applied. If yes go to c, else go to d
    c) modify SIP message by applying delete
    d) determine if insert should be applied. If yes go to e, else go to f
    e) modify SIP message by applying insert
    f) determine if replace should be applied. If yes go to g, else go to 2
    g) modify SIP message by applying replace
3. Finished. Return SIP message to the controller module.

Each modification that a modifier does to a SIP message is stored together with the SIP message. By storing modifications, the modifier module has the ability to undo them at a later stage.

Rules that modifies the actual content of SIP messages can make use of the sipMsgParser object to access the SIP message. It is a simple parser that treats SIP messages as an array of characters. Since the content of a SIP message is likely to be modified several times, the sipMsgParser object re-parses the SIP message every time it is called.

6.3.2 Demodifying SIP messages

The modifier module has the ability to, given a SIP message that it already has modified, undo the modifications. It is possible since the modifications that has been done to a SIP message are stored together with the message. It is the demodifier object that is responsible for removing modifications.

Note that the modifier module is very restricted when undoing modifications since each modification is likely to depend on earlier modifications. The ability to remove modifications from a SIP message is not a necessary requirement in the context of this thesis. Therefore, no further details will be given on the topic. See Lundin (2007) for additional details.

6.4 Container module

The container module stores the SIP messages that passes through the test tool. The capacity of the container is set by the user prior to the execution of the test tool. If the capacity is exceeded during the execution of the test tool, SIP messages will be overwritten in a first in, first out (FIFO) fashion. The container module was not a part of this Master's project and is therefore not described in any further detail. See Lundin (2007) for more details. See appendix A for the module's class diagram.

6.5 SUT failure handler module

The SUT failure handler module is responsible for monitoring the subject under test (SUT) while the test tool is running, in order to detect software failures in the SUT. In case a failure is detected, the SUT failure handler module alerts the controller module of it. To accomplish this, the module runs in a separate thread, in parallel with the main thread (which stores, modifies and sends SIP messages until it is interrupted by a signal from the SUT failure handler module).

The module is also responsible for storing information about the software failures it detects. By doing so the SUT failure handler module can, when it detects a failure, consult the previously stored information and draw conclusions about the character of the failure. The module can for example conclude if the failure has occured before, if the test tool already has been able to determine the reason for the failure, and so on. When it then signals that a failure has occured, the conclusions about the character of the failure are sent along with the signal. The controller module,

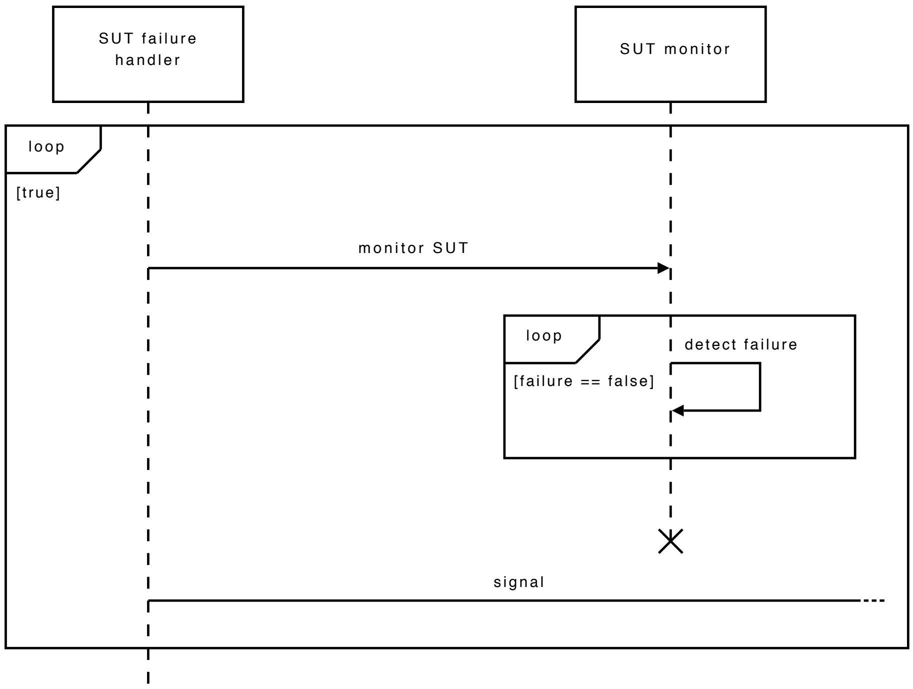


Figure 6.5. As long as the test tool is running, the SUT failure handler module moitors the SUT. If a failure is detected, the SUT failure handler module interrupts the test tool's main thread with a signal and then continues to monitor the SUT.

which receives the signal, is then able to take appropriate action. The module, which consists of a single object, detects failures by launching a separate application, which is referred to as the SUT monitor. The typical scenario can be seen in figure 6.5.

How the SUT monitor detects software failures depends on how the SUT is implemented. If for example the SUT writes software failures to a log file, the SUT monitor can detect failures by monitoring the log file. The SUT failure handler module was developed by both me and Carl Lundin. An alternative description of this module can therefore be seen in Lundin (2007). See appendix A for the module's class diagram.

6.6 Transporter module

The transporter modules main responsibility is to receive incoming, and send outgoing, SIP messages. The transporter module is capable of transporting messages over both UDP (connectionless transport layer protocol) and TCP (connection-oriented transport layer protocol). The objects that the transporter module consists of can be seen in table 6.2. See appendix A for the module's class diagram.

Object	Purpose
receiver	Receive incoming SIP messages.
sender	Send outgoing SIP messages.
transformer	Transform incoming SIP messages into a fleshed out format that the rest of the test tool expects. Transform outgoing SIP messages into the standard SIP format.
trafficFilter	Stop irrelevant SIP messages that are received by the transporter module from reaching any other part of the test tool.

Table 6.2. The fundamental objects that the transporter module consists of. Every object is an instance of a C++ class.

6.6.1 Receiving SIP messages

SIP elements which sends SIP messages to the test tool connects to the receiver object. In order for the test tool to handle the routing of received SIP messages as described in section 6.1.2, different SIP elements that connects to the receiver may have to connect to different addresses.

The receiver object handles this by allowing connections on several transport layer addresses in parallel. It accomplishes this by being multithreaded, two threads responsible for handling connections on a specific address. It is two threads per address since the transporter module is capable of handling both UDP and TCP connections on each specific address.

As a result of the multithreaded receiver object, several SIP messages can be received in parallel. Each thread that receives a SIP message places it in a queue shared by all the threads. The queue is a shared resource and semaphores [Tannenbaum, 2001] are used to restrict access to it. The queue is shared because the test tool tries not to disrupt the order in which the SIP messages would have been sent had the test tool not been there.

The main thread will remove SIP messages from the queue in a first in, first out (FIFO) fashion. A message that is removed from the queue is handled by the rest of the test tool before another message is removed from the queue. Figure 6.6 shows how the the objects that constitutes the transporter module handles a received SIP message. The receiver object removes a SIP message from the queue containing received SIP messages. It passes the SIP message to the transformer object, which changes the format of the SIP message into an internal format that the rest of the test tool works with. The internal format is able to carry information about the IP packet that the SIP message was received in and information about modifications that are made to a SIP message. The transformer attaches the source and destination addresses from the IP packet that contained the received SIP message to the SIP message.

The SIP message is then passed to the trafficFilter object, which in turn just

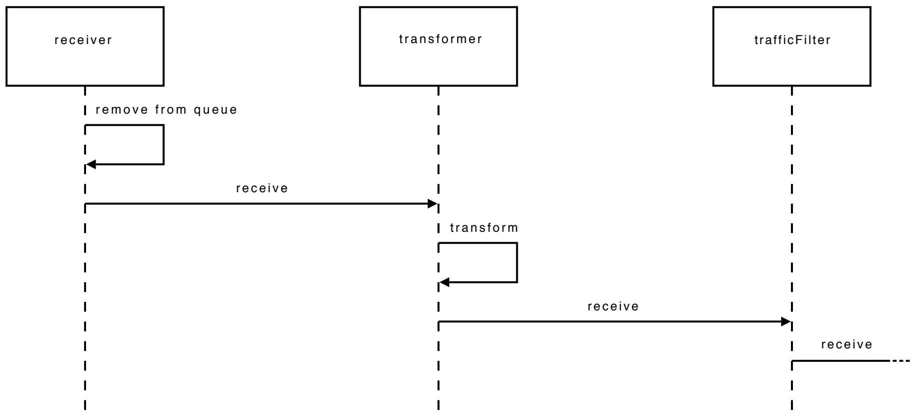


Figure 6.6. The transporter module processes a SIP message received from another SIP element. The message is removed from the queue containing received SIP messages and, when the transporter module is finished processing it, passed on to the controller module.

forwards it to the system controller module. The trafficFilter object has the ability to silently discard SIP messages that are judged to be irrelevant. Note that the trafficFilter object has no relevant purpose in this Master's project. All received SIP messages are considered as relevant when the test tool is operating in the mode where it provokes software failures. The trafficFilter object will therefore not be described in any further detail in this thesis. See Lundin (2007) for a description of its purpose.

6.6.2 Sending SIP messages

Figure 6.7 shows how the transporter module sends a SIP message, which it has received from the controller module, to another SIP element. The traffic filter object does not have any function when the transporter module is sending SIP messages. It just forwards any SIP messages it receives to the transformer object. The transformer changes the format of the SIP message to the format that the sender uses. It then passes the SIP message, together with information concerning where and how it should be sent, to the sender. Finally, the sender sends the SIP message. Note that a SIP message will, as default, be sent with the same transport layer protocol that it was received with.

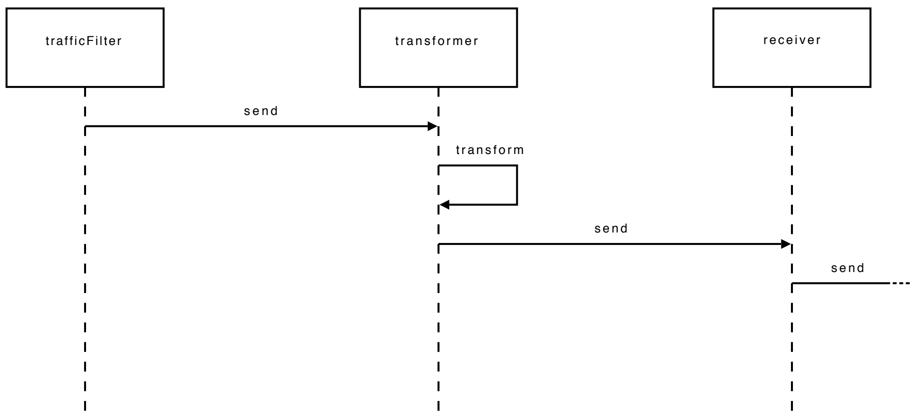


Figure 6.7. The transporter module processes a SIP message received from the controller module.

Chapter 7

Evaluation of the test tool

In this chapter three environments in which the test tool has been executed are described. The behaviour of the test tool in each and one of them is evaluated. The mode in which the test tool attempts to provoke software failures in the subject under test (SUT) is the mode that will be evaluated. See Lundin (2007) for an evaluation of the mode in which the test tool determines the reason for a provoked software failure.

7.1 Testing a simulated SIP proxy server

The environment, in which the test tool treats a simulated SIP proxy server as the SUT, consists of the simulated SIP proxy server, the test tool and two SIP user agents (SleIPner A and SleIPner B). The SIP proxy server is simulated in the sense that it never examines the content of SIP messages that it receives in order to route them correctly. Instead, it routes SIP messages in the same way that the test tool does (see section 6.1.2 for a description). One way to accomplish a correct routing in the just described environment can be seen in figure 7.1.

The test tool is in the described environment capable of receiving, routing and sending SIP messages as it should. SIP messages that the test tool receives from SleIPner A and SleIPner B are modified, in order to be made unexpected, and sent to the SUT. It can tell the difference between SleIPner A and SleIPner B, and is therefore able to modify SIP messages received from one of them different than those received from the other. The test tool is also capable of detecting simulated SUT software failures.

The failures are simulated because it is not possible to provoke software failures in the SUT since it does not examine the content of received SIP messages. Instead, the SUT simulates a software failure after receiving a, from the test tool's point of view unknown, number of SIP messages. The SUT simulates a software failure by writing to a log file. The test tool then detects software failures by reading the log file. This environment was used throughout the development of the test tool in order to test the test tool itself.

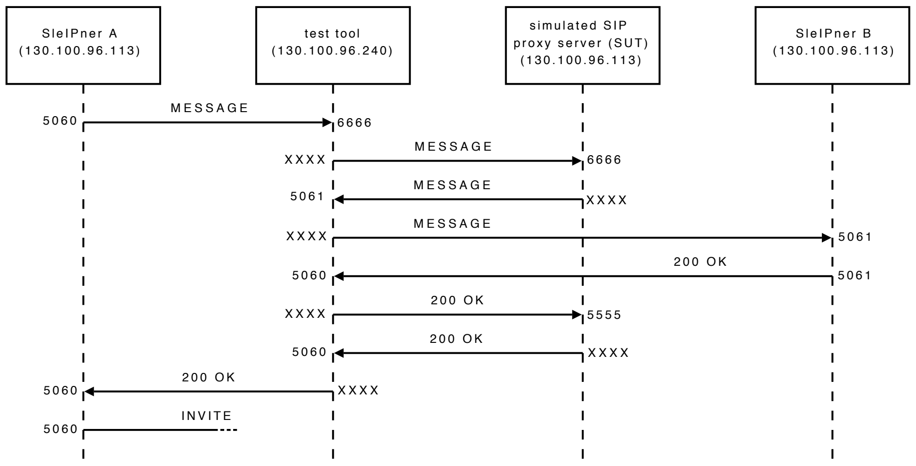

SleIPner A (130.100.96.113)
test tool (130.100.96.240)
simulated SIP proxy server (SUT) (130.100.96.113)
SleIPner B (130.100.96.113)
MESSAGE
5060
6666
XXXX
5061
200 OK
5555
INVITE

Figure 7.1. An environment in which the test tool was executed: the simulated SIP proxy server is the subject under test (SUT). SleIPner A and SleIPner B are the SIP user agents that generates the SIP traffic that the test tool modifies and sends to the SUT. The numbers at the start and end of each arrow are the port numbers used to send the message.

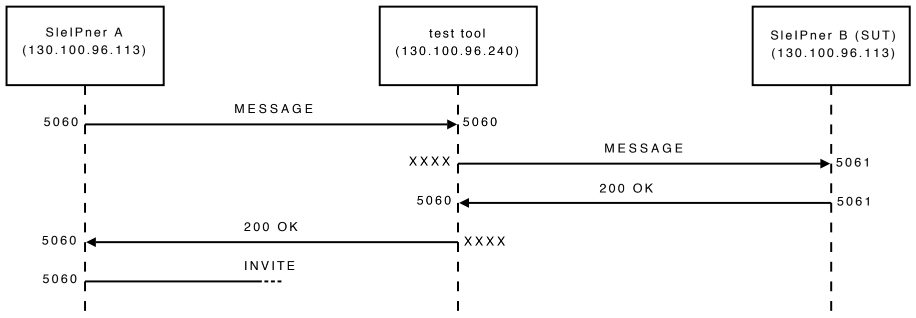

SleIPner A (130.100.96.113)
test tool (130.100.96.240)
SleIPner B (SUT) (130.100.96.113)
MESSAGE
5060
XXXX
5061
200 OK
INVITE

Figure 7.2. An environment in which the test tool was executed: SleIPner B is the SUT. SleIPner A is the user agent that generates the SIP traffic that the test tool modifies and sends to the SUT. The numbers at the start and end of each arrow are the port numbers used to send the message.

7.2 Testing a SIP user agent server

The environment, in which the test tool treats a SIP user agent server as the SUT, consists of two SIP user agents (SleIPner A and SleIPner B) and the test tool. SleIPner B is the SUT in the environment which can be seen 7.2. The test tool is in this environment capable of receiving, routing and sending SIP messages as it should. SIP messages that the test tool receives from SleIPner A are modified, in order to be made unexpected, and sent to the SUT. There has been no serious attempts in provoking software failures in SleIPner B. The main reason for this

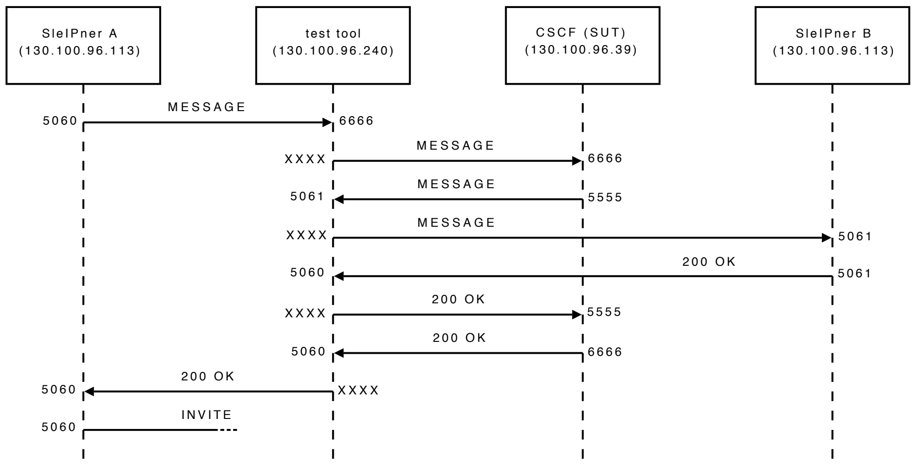


Figure 7.3. An environment in which the test tool was executed: the CSCF is the SUT. SleIPner A is the user agent that generates the SIP traffic that the test tool modifies and sends to the SUT. The numbers at the start and end of each arrow are the port numbers used to send the message.

was the lack of a SIP element capable of automatically creating and sending SIP messages to the test tool. The environment was instead mainly setup to ensure that the test tool is capable of treating a SIP user agent server as the SUT.

7.3 Testing a SIP proxy server

The environment, in which the test tool treats a SIP proxy server as the SUT, consists of the SIP proxy server (the CSCF), the test tool and two SIP user agents (SleIPner A and SleIPner B). Figure 7.3 depicts the described environment. In order to get the user agents and the CSCF to route SIP messages correctly in this environment, the test tool sees to that the CSCF is unaware of the user agents and the user agents unaware of the CSCF. The test tool modifies the content of the SIP messages it receives in order to accomplish this, focusing on SIP header fields containing route information (e.g. Via, From and To). This is not an optimal solution. The test tool must now, in order to work as intended, parse received SIP messages in order to find and modify the header fields containing route information. The solution is not optimal since the test tool is likely to receive malformed SIP messages and therefore may not be able to locate and modify the header fields containing route information.

SUT software failures have only been simulated in this environment. No serious attempts to provoke real SUT software failures have been made, they have been simulated instead. The main reason for this was the lack of a SIP element capable of automatically creating and sending SIP messages to the test tool.

Chapter 8

Conclusions and outlook

In this chapter conclusions are drawn and suggestions for further development and usage of the tool is suggested.

8.1 Conclusions

We have seen how a tool for testing the robustness in arbitrary SIP servers can be designed. The requirements that Ericsson AB had on the tool, stated in section 1.1 have been fulfilled and the problems that were formulated as questions in section 1.2 have been investigated and answered.

By feeding the subject under test (SUT) with unexpected SIP input using the same interfaces as any other SIP element would use, the tool can treat the SUT as a black box. The tool uses the least common denominator, shared by all SIP servers, in order to inject unexpected SIP input into the SUT. As a consequence, the tool can inject unexpected input into arbitrary SIP servers. This solution was straightforward and so far no disadvantages with it, or better solutions, have been discovered.

It can be discussed whether the decision to rely on other SIP elements in the generation of unexpected SIP input was a good decision. A major advantage with this decision is that the tool in theory has unlimited access to the whole set of SIP messages that the SUT can ever expect to receive. In order for the tool to have that access without relying on other SIP elements, it would have to be able to mimic all possibe SIP elements. To implement such a tool was judged as impossible, especially considering that time is a limited resource in a Master's project. The major disadvantage with this method is that the tool has to function more or less like a SIP proxy server since also the SUT, as mentioned in section 5.2, will send SIP messages to the tool. As a consequence, not all SIP messages that the tool receives are supposed to be sent to the SUT and the tool has to, just as a SIP proxy server, route received SIP messages.

The decision to route SIP messages without examining their content, as also was mentioned in section 5.2, delimits the set of environments in which the tool is able to operate. It also makes the tool considerably harder to configure in order to get it to route SIP messages correctly. However, it is still considered to be the right decision since the tool itself is expected to receive a great deal of invalid SIP messages. If the tool was to be able to examine these invalid SIP messages, it would have to be more robust than the SUT, which clearly is not a requirement that the tool should have to fulfill. Unfortunately, the tool had to examine and modify the content of SIP messages in order for the routing to be correct in the environment described in section 7.3. An efficient solution to this routing problem is yet to be found.

The fact that the tool has yet to provoke any real software failures is not regarded as a weakness in the tool, since no serious attempts to provoke real software failures have been made. As mentioned in section 5.1, the test tool will in general have to send a large amount of SIP messages to the SUT in order to provoke a software failure. The environments in which the tool has been executed so far has lacked elements capable of generating a large number of SIP messages, that the SUT expects, automatically. Note that such elements do exist.

It can be concluded that it is possible to develop a fully automated tool, capable of testing the robustness in different types of SIP servers. The tool, which in theory basically has no limitations, is somewhat weakened by the many routing mechanisms of SIP. Although weakened, the tool should still be very useful when testing the robustness of SIP servers.

8.2 Outlook

The natural step from here would be to let the tool attempt to provoke real software failures. This can be achieved by configuring SIP elements, capable of generating a large number of SIP messages that the SUT expects automatically, to send their output, with the SUT as the target, through the test tool. This step should be small, since SIP elements capable of generating large numbers of SIP messages automatically exists and are used at Ericsson AB.

The problem with the routing of SIP messages, that was experienced with the CSCF as SUT, also needs further investigation. The current "work-around" solution is, as mentioned in 7.3, not an optimal solution since it is likely to expose not only the SUT but also the tool itself to unexpected SIP input. A solution in which the tool does not have to examine the content of received SIP messages would therefore be preferred. With the current solution, the parser that the tool uses to access the content of received SIP messages needs improvement.

One possible improvement of the tool would be to refactor or replace the transporter module, focusing on the receiver and the sender objects. For the moment they are both implemented in a very straightforward way, not optimized and not as robust as they could be.

If the tool is experienced as slow, a possible optimization could be to replace the C++ strings with C character arrays. The fact that C++ strings are slower to work with than C character arrays together with the fact that the tool uses C++ strings frequently suggests that it should be a fruitful optimization.

Bibliography

[Beizer, 1995] Beizer, B. 1995. *Black-Box Testing, Techniques for Functional Testing of Software and Systems*. John Wiley & Sons Ltd. USA. 294 pages. ISBN: 0-471-12094-4.

[Bieman and Yin, 1992] Bieman, J., Yin, H. 1992. *Designing For Software Testability Using Automated Oracles*. Proceedings of International Test Conference. Pages 900-907.

[Camarillo, 2002] Camarillo, G. 2002. *SIP Demystified*. McGraw-Hill. USA. 264 pages. ISBN: 0-07-137340-3.

[Camarillo and García-Martín, 2006] Camarillo, G. and García-Martín, M. 2006. *The 3G IP Multimedia Subsystem (IMS)*. John Wiley & Sons Ltd. Chippenham, England. 2nd edition. 427 pages. ISBN-13 978-0-470-01818-7.

[Campbell et al., 2002] Campbell, B., Rosenberg, J., Schulzrinne, H., Huitema, C. and Gurle, D. 2002. *RFC 3428. Session Initiation Protocol (SIP) Extension for Instant Messaging*. Network Working Group. 18 pages.

[Lundin, 2007] Lundin, C. 2007. *Determining Reasons for Software Failures in SIP servers – the Development of an Automatic Test Tool*. Master's thesis at CSC, KTH.

[Mankefors et al., 2003] Mankefors, S., Torkar, R., Boklund, A. 2003. *New Quality Estimations in Random Testing*. Proceedings of the 14th International Symposium on Software Reliability Engineering. Pages 468–478.

[Myers et al., 2004] Myers, G., Badgett, T., Thomas, T. and Sandler, C. 2004. *The Art of Software Testing*. John Wiley & Sons, Inc. 2nd edition. USA. 234 pages. ISBN: 0-471-46912-2.

[Rosenberg et al., 2002] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and Schooler, E. 2002. *RFC 3261. SIP: Session Initiation Protocol*. Network Working Group. 269 pages.

[Sommerville, 2004] Sommerville, I. 2004. *Software Engineering*. Addison Wesley. 7th edition. USA. 759 pages. ISBN: 0-321-21026-3.

[Tannenbaum, 2001] Tannenbaum, A. 2001. *Modern Operating Systems.* Prentice Hall. 2nd edition. USA. 976 pages. ISBN: 0130313580.

[Wieser et al., 2004] Wieser, C., Laakso, M. and Schulzrinne, H. 2004. *Security Testing of SIP Implementations.* SOQUA/TECOS. Pages 165–178

Appendix A

Class diagrams

In this appendice five class diagrams are depicted – one diagram for every module that the test tool constitutes of.

A.1 Controller module

The class diagram belonging to the controller module, which is described in section 6.2 can be seen in figure A.1 (on the next page).

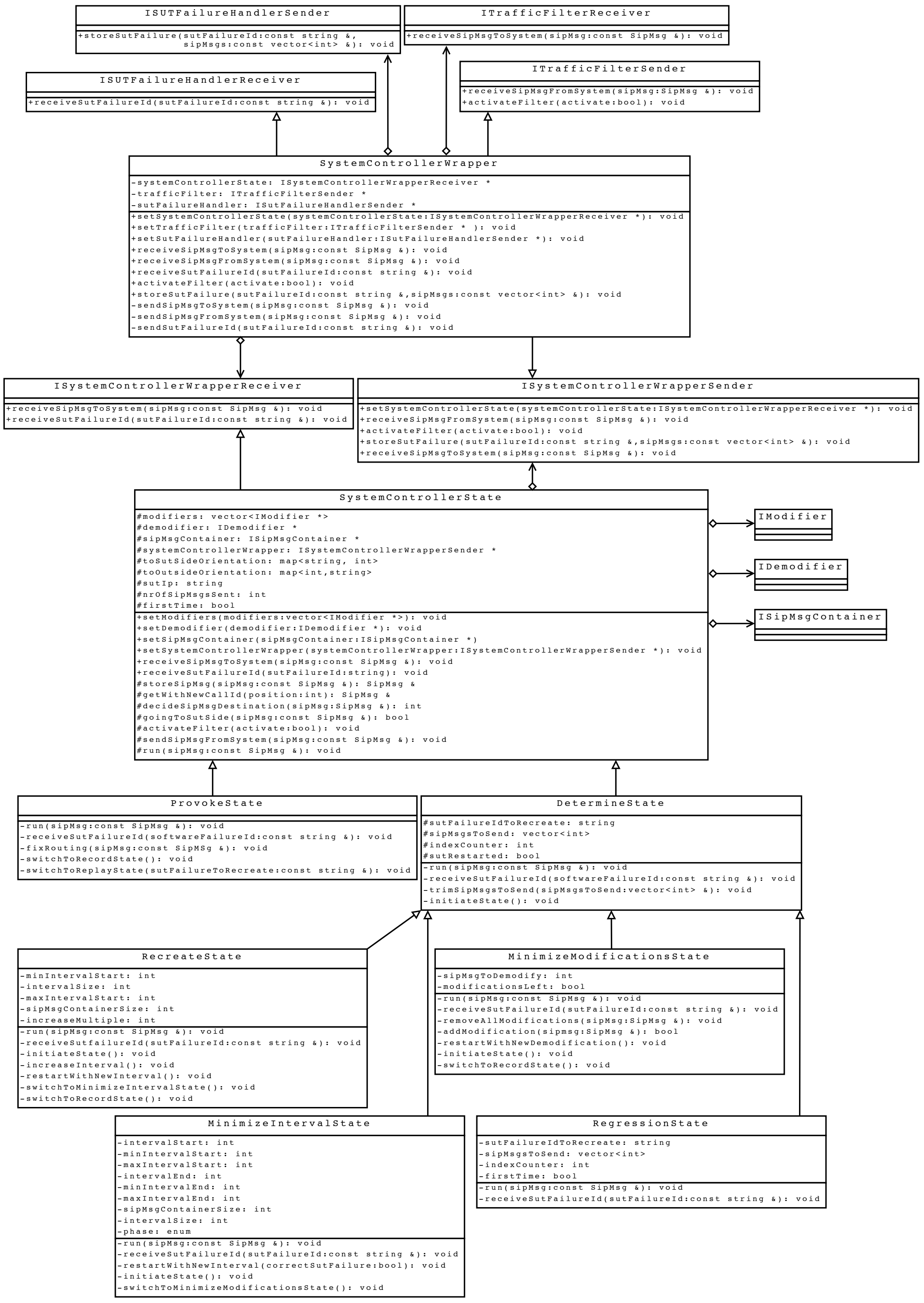

ISUTFailureHandlerSender
+storeSutFailure(sutFailureId:const string &, sipMsgs:const vector<int> &): void
ITrafficFilterReceiver
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
ITrafficFilterSender
+receiveSipMsgFromSystem(sipMsg:SipMsg &): void
+activateFilter(activate:bool): void
ISUTFailureHandlerReceiver
+receiveSutFailureId(sutFailureId:const string &): void
SystemControllerWrapper
-systemControllerState: ISystemControllerWrapperReceiver *
-trafficFilter: ITrafficFilterSender *
-sutFailureHandler: ISutFailureHandlerSender *
+setSystemControllerState(systemControllerState:ISystemControllerWrapperReceiver *): void
+setTrafficFilter(trafficFilter:ITrafficFilterSender *): void
+setSutFailureHandler(sutFailureHandler:ISutFailureHandlerSender *): void
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
+receiveSipMsgFromSystem(sipMsg:const SipMsg &): void
+receiveSutFailureId(sutFailureId:const string &): void
+activateFilter(activate:bool): void
+storeSutFailure(sutFailureId:const string &,sipMsgs:const vector<int> &): void
-sendSipMsgToSystem(sipMsg:const SipMsg &): void
-sendSipMsgFromSystem(sipMsg:const SipMsg &): void
-sendSutFailureId(sutFailureId:const string &): void
ISystemControllerWrapperReceiver
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
+receiveSutFailureId(sutFailureId:const string &): void
ISystemControllerWrapperSender
+setSystemControllerState(systemControllerState:ISystemControllerWrapperReceiver *): void
+receiveSipMsgFromSystem(sipMsg:const SipMsg &): void
+activateFilter(activate:bool): void
+storeSutFailure(sutFailureId:const string &,sipMsgs:const vector<int> &): void
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
SystemControllerState
#modifiers: vector<IModifier *>
#demodifier: IDemodifier *
#sipMsgContainer: ISipMsgContainer *
#systemControllerWrapper: ISystemControllerWrapperSender *
#toSutSideOrientation: map<string, int>
#toOutsideOrientation: map<int,string>
#sutIp: string
#nrOfSipMsgsSent: int
#firstTime: bool
+setModifiers(modifiers:vector<IModifier *>): void
+setDemodifier(demodifier:IDemodifier *): void
+setSipMsgContainer(sipMsgContainer:ISipMsgContainer *)
+setSystemControllerWrapper(systemControllerWrapper:ISystemControllerWrapperSender *): void
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
+receiveSutFailureId(sutFailureId:string): void
#storeSipMsg(sipMsg:const SipMsg &): SipMsg &
#getWithNewCallId(position:int): SipMsg &
#decideSipMsgDestination(sipMsg:SipMsg &): int
#goingToSutSide(sipMsg:const SipMsg &): bool
#activateFilter(activate:bool): void
#sendSipMsgFromSystem(sipMsg:const SipMsg &): void
#run(sipMsg:const SipMsg &): void
IModifier
IDemodifier
ISipMsgContainer
ProvokeState
-run(sipMsg:const SipMsg &): void
-receiveSutFailureId(softwareFailureId:const string &): void
-fixRouting(sipMsg:const SipMSg &): void
-switchToRecordState(): void
-switchToReplayState(sutFailureToRecreate:const string &): void
DetermineState
#sutFailureIdToRecreate: string
#sipMsgsToSend: vector<int>
#indexCounter: int
#sutRestarted: bool
-run(sipMsg:const SipMsg &): void
-receiveSutFailureId(softwareFailureId:const string &): void
-trimSipMsgsToSend(sipMsgsToSend:vector<int> &): void
-initiateState(): void
RecreateState
-minIntervalStart: int
-intervalSize: int
-maxIntervalStart: int
-sipMsgContainerSize: int
-increaseMultiple: int
-run(sipMsg:const SipMsg &): void
-receiveSutfailureId(sutFailureId:const string &): void
-initiateState(): void
-increaseInterval(): void
-restartWithNewInterval(): void
-switchToMinimizeIntervalState(): void
-switchToRecordState(): void
MinimizeModificationsState
-sipMsgToDemodify: int
-modificationsLeft: bool
-run(sipMsg:const SipMsg &): void
-receiveSutFailureId(sutFailureId:const string &): void
-removeAllModifications(sipMsg:SipMsg &): void
-addModification(sipmsg:SipMsg &): bool
-restartWithNewDemodification(): void
-initiateState(): void
-switchToRecordState(): void
MinimizeIntervalState
-intervalStart: int
-minIntervalStart: int
-maxIntervalStart: int
-intervalEnd: int
-minIntervalEnd: int
-maxIntervalEnd: int
-sipMsgContainerSize: int
-intervalSize: int
-phase: enum
-run(sipMsg:const SipMsg &): void
-receiveSutFailureId(sutFailureId:const string &): void
-restartWithNewInterval(correctSutFailure:bool): void
-initiateState(): void
-switchToMinimizeModificationsState(): void
RegressionState
-sutFailureIdToRecreate: string
-sipMsgsToSend: vector<int>
-indexCounter: int
-firstTime: bool
-run(sipMsg:const SipMsg &): void
-receiveSutFailureId(sutFailureId:const string &): void

Figure A.1. The class diagram belonging to the controller module.

A.2 Modifier module

The class diagram belonging to the modifier module, which is described in section 6.3 can be seen in figure A.2.

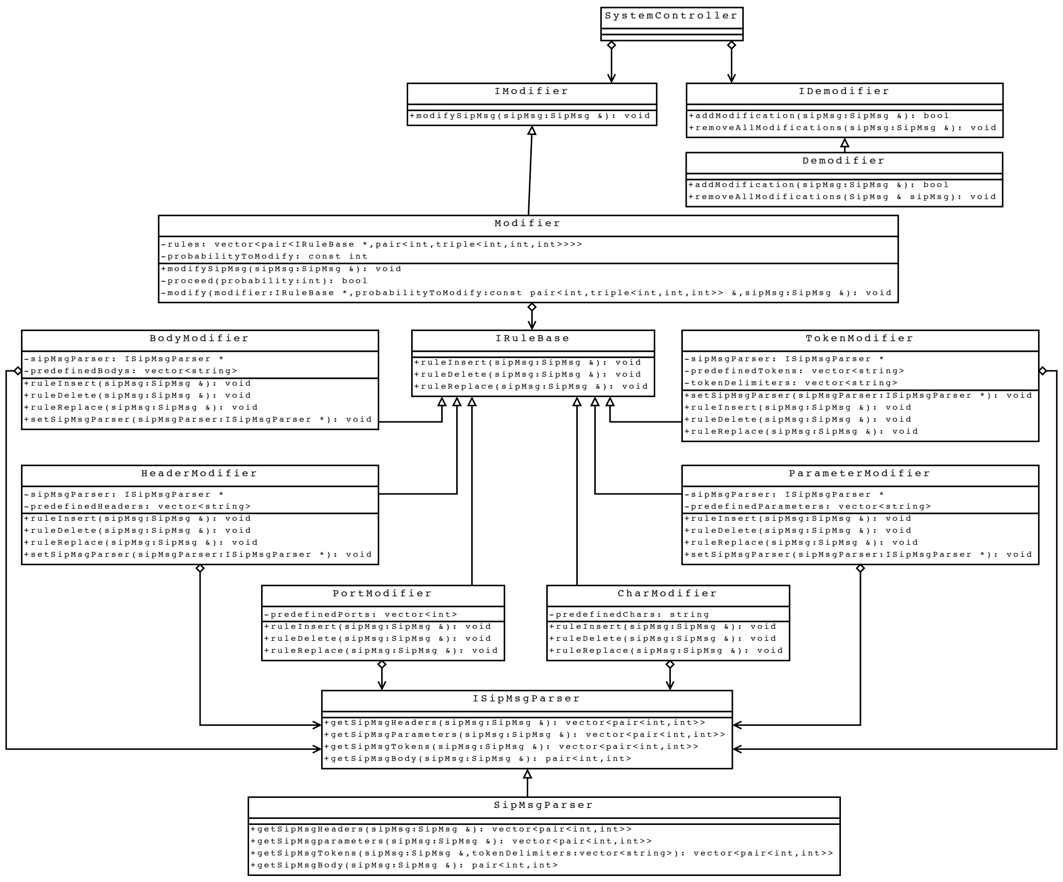

SystemController
IModifier
+modifySipMsg(sipMsg:SipMsg &): void
IDemodifier
+addModification(sipMsg:SipMsg &): bool
+removeAllModifications(sipMsg:SipMsg &): void
Demodifier
+addModification(sipMsg:SipMsg &): bool
+removeAllModifications(SipMsg & sipMsg): void
Modifier
-rules: vector<pair<IRuleBase *,pair<int,triple<int,int,int>>>>
-probabilityToModify: const int
+modifySipMsg(sipMsg:SipMsg &): void
-proceed(probability:int): bool
-modify(modifier:IRuleBase *,probabilityToModify:const pair<int,triple<int,int,int>> &,sipMsg:SipMsg &): void
BodyModifier
-sipMsgParser: ISipMsgParser *
-predefinedBodys: vector<string>
+ruleInsert(sipMsg:SipMsg &): void
+ruleDelete(sipMsg:SipMsg &): void
+ruleReplace(sipMsg:SipMsg &): void
+setSipMsgParser(sipMsgParser:ISipMsgParser *): void
IRuleBase
+ruleInsert(sipMsg:SipMsg &): void
+ruleDelete(sipMsg:SipMsg &): void
+ruleReplace(sipMsg:SipMsg &): void
TokenModifier
-sipMsgParser: ISipMsgParser *
-predefinedTokens: vector<string>
-tokenDelimiters: vector<string>
+setSipMsgParser(sipMsgParser:ISipMsgParser *): void
+ruleInsert(sipMsg:SipMsg &): void
+ruleDelete(sipMsg:SipMsg &): void
+ruleReplace(sipMsg:SipMsg &): void
HeaderModifier
-sipMsgParser: ISipMsgParser *
-predefinedHeaders: vector<string>
+ruleInsert(sipMsg:SipMsg &): void
+ruleDelete(sipMsg:SipMsg &): void
+ruleReplace(sipMsg:SipMsg &): void
+setSipMsgParser(sipMsgParser:ISipMsgParser *): void
ParameterModifier
-sipMsgParser: ISipMsgParser *
-predefinedParameters: vector<string>
+ruleInsert(sipMsg:SipMsg &): void
+ruleDelete(sipMsg:SipMsg &): void
+ruleReplace(sipMsg:SipMsg &): void
+setSipMsgParser(sipMsgParser:ISipMsgParser *): void
PortModifier
-predefinedPorts: vector<int>
+ruleInsert(sipMsg:SipMsg &): void
+ruleDelete(sipMsg:SipMsg &): void
+ruleReplace(sipMsg:SipMsg &): void
CharModifier
-predefinedChars: string
+ruleInsert(sipMsg:SipMsg &): void
+ruleDelete(sipMsg:SipMsg &): void
+ruleReplace(sipMsg:SipMsg &): void
ISipMsgParser
+getSipMsgHeaders(sipMsg:SipMsg &): vector<pair<int,int>>
+getSipMsgParameters(sipMsg:SipMsg &): vector<pair<int,int>>
+getSipMsgTokens(sipMsg:SipMsg &): vector<pair<int,int>>
+getSipMsgBody(sipMsg:SipMsg &): pair<int,int>
SipMsgParser
+getSipMsgHeaders(sipMsg:SipMsg &): vector<pair<int,int>>
+getSipMsgparameters(sipMsg:SipMsg &): vector<pair<int,int>>
+getSipMsgTokens(sipMsg:SipMsg &,tokenDelimiters:vector<string>): vector<pair<int,int>>
+getSipMsgBody(sipMsg:SipMsg &): pair<int,int>

Figure A.2. The class diagram belonging to the modifier module.

A.3 Container module

The class diagram belonging to the container module, which is described in section 6.4 can be seen in figure A.3.

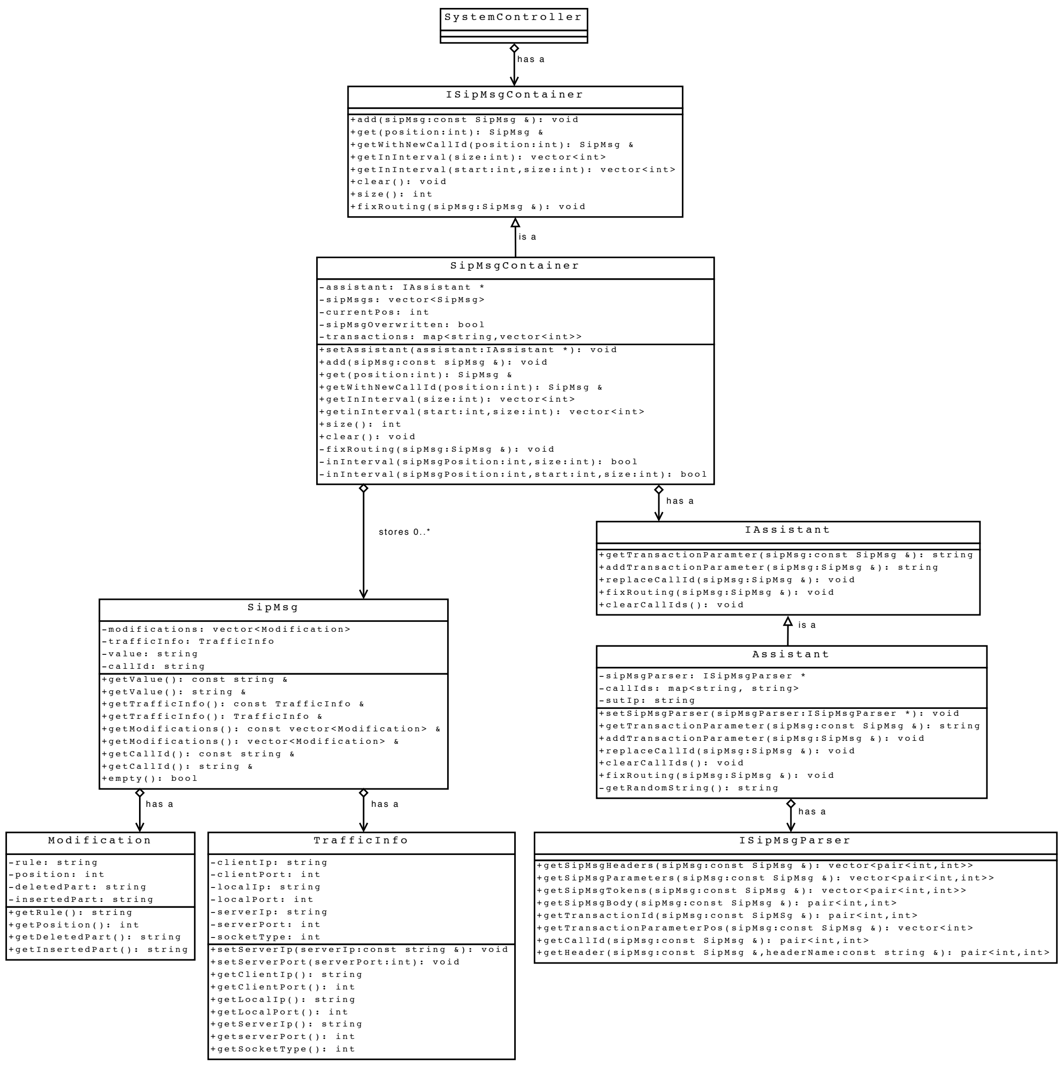

SystemController
has a
ISipMsgContainer
+add(sipMsg:const SipMsg &): void
+get(position:int): SipMsg &
+getWithNewCallId(position:int): SipMsg &
+getInInterval(size:int): vector<int>
+getInInterval(start:int,size:int): vector<int>
+clear(): void
+size(): int
+fixRouting(sipMsg:SipMsg &): void
is a
SipMsgContainer
-assistant: IAssistant *
-sipMsgs: vector<SipMsg>
-currentPos: int
-sipMsgOverwritten: bool
-transactions: map<string,vector<int>>
+setAssistant(assistant:IAssistant *): void
+add(sipMsg:const sipMsg &): void
+get(position:int): SipMsg &
+getWithNewCallId(position:int): SipMsg &
+getInInterval(size:int): vector<int>
+getinInterval(start:int,size:int): vector<int>
+size(): int
+clear(): void
-fixRouting(sipMsg:SipMsg &): void
-inInterval(sipMsgPosition:int,size:int): bool
-inInterval(sipMsgPosition:int,start:int,size:int): bool
stores 0..*
has a
IAssistant
+getTransactionParamter(sipMsg:const SipMsg &): string
+addTransactionParameter(sipMsg:SipMsg &): string
+replaceCallId(sipMsg:SipMsg &): void
+fixRouting(sipMsg:SipMsg &): void
+clearCallIds(): void
is a
Assistant
-sipMsgParser: ISipMsgParser *
-callIds: map<string, string>
-sutIp: string
+setSipMsgParser(sipMsgParser:ISipMsgParser *): void
+getTransactionParameter(sipMsg:const SipMsg &): string
+addTransactionParameter(sipMsg:SipMsg &): void
+replaceCallId(sipMsg:SipMsg &): void
+clearCallIds(): void
+fixRouting(sipMsg:SipMsg &): void
-getRandomString(): string
has a
ISipMsgParser
+getSipMsgHeaders(sipMsg:const SipMsg &): vector<pair<int,int>>
+getSipMsgParameters(sipMsg:const SipMsg &): vector<pair<int,int>>
+getSipMsgTokens(sipMsg:const SipMsg &): vector<pair<int,int>>
+getSipMsgBody(sipMsg:const SipMsg &): pair<int,int>
+getTransactionId(sipMsg:const SipMSg &): pair<int,int>
+getTransactionParameterPos(sipMsg:const SipMsg &): vector<int>
+getCallId(sipMsg:const SipMsg &): pair<int,int>
+getHeader(sipMsg:const SipMsg &,headerName:const string &): pair<int,int>
SipMsg
-modifications: vector<Modification>
-trafficInfo: TrafficInfo
-value: string
-callId: string
+getValue(): const string &
+getValue(): string &
+getTrafficInfo(): const TrafficInfo &
+getTrafficInfo(): TrafficInfo &
+getModifications(): const vector<Modification> &
+getModifications(): vector<Modification> &
+getCallId(): const string &
+getCallId(): string &
+empty(): bool
has a
has a
Modification
-rule: string
-position: int
-deletedPart: string
-insertedPart: string
+getRule(): string
+getPosition(): int
+getDeletedPart(): string
+getInsertedPart(): string
TrafficInfo
-clientIp: string
-clientPort: int
-localIp: string
-localPort: int
-serverIp: string
-serverPort: int
-socketType: int
+setServerIp(serverIp:const string &): void
+setServerPort(serverPort:int): void
+getClientIp(): string
+getClientPort(): int
+getLocalIp(): string
+getLocalPort(): int
+getServerIp(): string
+getserverPort(): int
+getSocketType(): int

Figure A.3. The class diagram belonging to the container module.

A.4 SUT failure handler module

The class diagram belonging to the SUT failure handler module, which is described in section 6.5 can be seen in figure A.4.

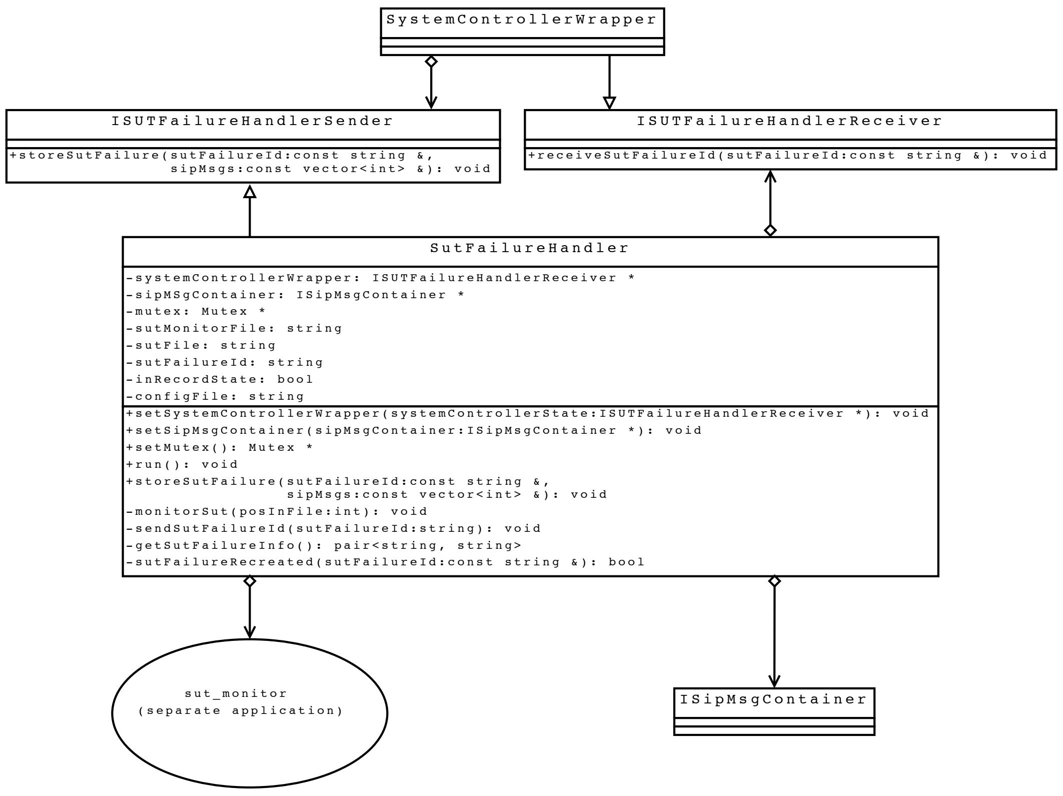

SystemControllerWrapper
ISUTFailureHandlerSender
+storeSutFailure(sutFailureId:const string &, sipMsgs:const vector<int> &): void
ISUTFailureHandlerReceiver
+receiveSutFailureId(sutFailureId:const string &): void
SutFailureHandler
-systemControllerWrapper: ISUTFailureHandlerReceiver *
-sipMSgContainer: ISipMsgContainer *
-mutex: Mutex *
-sutMonitorFile: string
-sutFile: string
-sutFailureId: string
-inRecordState: bool
-configFile: string
+setSystemControllerWrapper(systemControllerState:ISUTFailureHandlerReceiver *): void
+setSipMsgContainer(sipMsgContainer:ISipMsgContainer *): void
+setMutex(): Mutex *
+run(): void
+storeSutFailure(sutFailureId:const string &, sipMsgs:const vector<int> &): void
-monitorSut(posInFile:int): void
-sendSutFailureId(sutFailureId:string): void
-getSutFailureInfo(): pair<string, string>
-sutFailureRecreated(sutFailureId:const string &): bool
sut_monitor
(separate application)
ISipMsgContainer

Figure A.4. The class diagram belonging to the SUT failure handler module.

A.5 Transporter module

The class diagram belonging to the transporter module, which is described in section 6.6 can be seen in figure A.5.

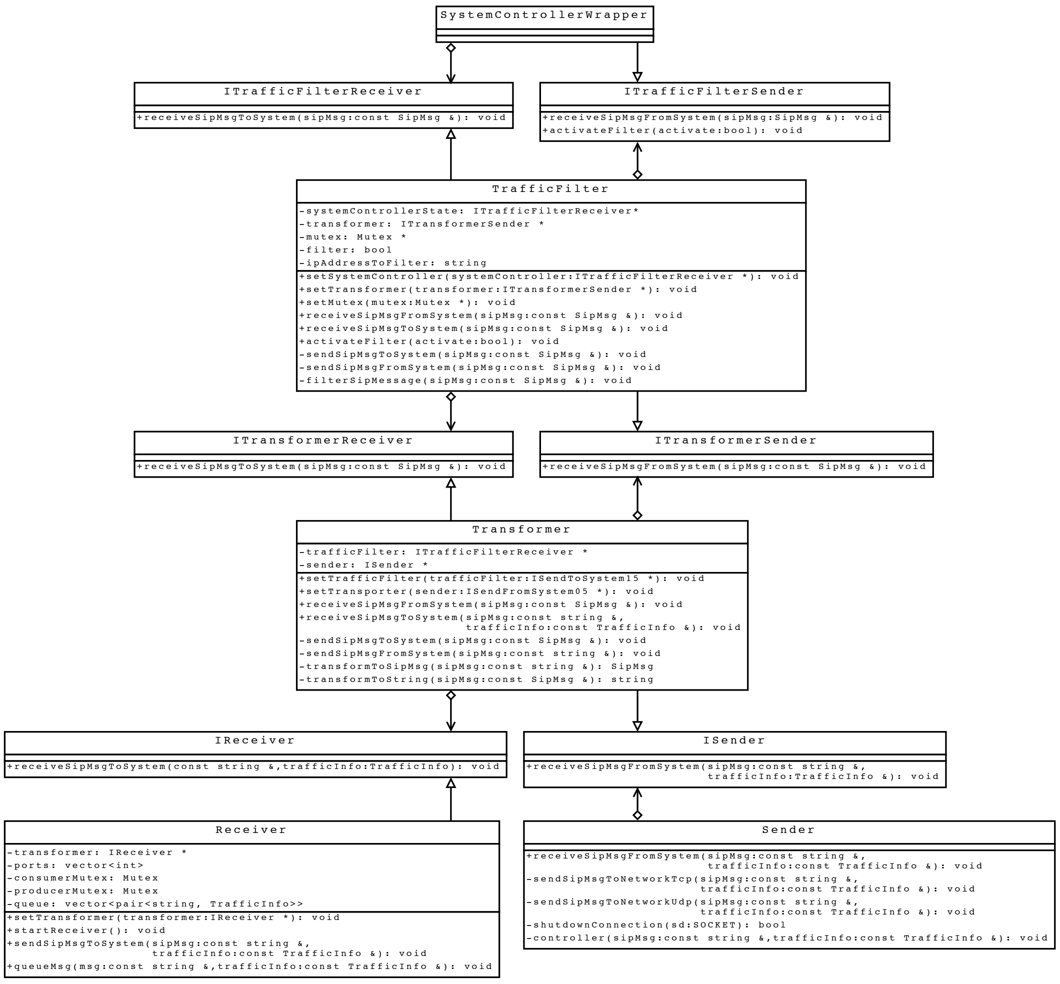

SystemControllerWrapper
ITrafficFilterReceiver
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
ITrafficFilterSender
+receiveSipMsgFromSystem(sipMsg:SipMsg &): void
+activateFilter(activate:bool): void
TrafficFilter
-systemControllerState: ITrafficFilterReceiver*
-transformer: ITransformerSender *
-mutex: Mutex *
-filter: bool
-ipAddressToFilter: string
+setSystemController(systemController:ITrafficFilterReceiver *): void
+setTransformer(transformer:ITransformerSender *): void
+setMutex(mutex:Mutex *): void
+receiveSipMsgFromSystem(sipMsg:const SipMsg &): void
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
+activateFilter(activate:bool): void
-sendSipMsgToSystem(sipMsg:const SipMsg &): void
-sendSipMsgFromSystem(sipMsg:const SipMsg &): void
-filterSipMessage(sipMsg:const SipMsg &): void
ITransformerReceiver
+receiveSipMsgToSystem(sipMsg:const SipMsg &): void
ITransformerSender
+receiveSipMsgFromSystem(sipMsg:const SipMsg &): void
Transformer
-trafficFilter: ITrafficFilterReceiver *
-sender: ISender *
+setTrafficFilter(trafficFilter:ISendToSystem15 *): void
+setTransporter(sender:ISendFromSystem05 *): void
+receiveSipMsgFromSystem(sipMsg:const SipMsg &): void
+receiveSipMsgToSystem(sipMsg:const string &, trafficInfo:const TrafficInfo &): void
-sendSipMsgToSystem(sipMsg:const SipMsg &): void
-sendSipMsgFromSystem(sipMsg:const string &): void
-transformToSipMsg(sipMsg:const string &): SipMsg
-transformToString(sipMsg:const SipMsg &): string
IReceiver
+receiveSipMsgToSystem(const string &,trafficInfo:TrafficInfo): void
ISender
+receiveSipMsgFromSystem(sipMsg:const string &, trafficInfo:TrafficInfo &): void
Receiver
-transformer: IReceiver *
-ports: vector<int>
-consumerMutex: Mutex
-producerMutex: Mutex
-queue: vector<pair<string, TrafficInfo>>
+setTransformer(transformer:IReceiver *): void
+startReceiver(): void
+sendSipMsgToSystem(sipMsg:const string &, trafficInfo:const TrafficInfo &): void
+queueMsg(msg:const string &,trafficInfo:const TrafficInfo &): void
Sender
+receiveSipMsgFromSystem(sipMsg:const string &, trafficInfo:const TrafficInfo &): void
-sendSipMsgToNetworkTcp(sipMsg:const string &, trafficInfo:const TrafficInfo &): void
-sendSipMsgToNetworkUdp(sipMsg:const string &, trafficInfo:const TrafficInfo &): void
-shutdownConnection(sd:SOCKET): bool
-controller(sipMsg:const string &,trafficInfo:const TrafficInfo &): void

Figure A.5. The class diagram belonging to the transporter module.