# Development of Nokia’s 5G testing tool

Expanding functionality to CP-E2 component

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor’s thesis

Information and communications technology, Riihimäki

Autumn, 2022

Leevi Räsänen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikka Tiivistelmä
Tekijä Leevi Räsänen Vuosi 2022
Työn nimi Nokian 5G testityökalun kehittäminen: Toiminnallisuuden laajentaminen CP-E2 komponentille
Ohjaaja Toni Laitinen

---

Nokia yrittää selvittää, onko Endeavour-nimisen työkalun avulla mahdollista parantaa 5G-ohjelmiston testausmenettelyä 5G-ohjaustasossa. Tällä hetkellä Endeavourin kehittämisen päätavoitteena on nähdä, voiko siitä tulla käyttökelpoinen testausväline. Osa tätä kelpoisuuden arviointia on tuen toteuttaminen uusille ohjaustason komponenteille, jotta niitä voidaan testata Endeavourissa. On kiinnostusta nähdä, voidaanko yhtä tällaista komponenttia, CP-E2:ta, tukea. Tässä työssä tutkitaan onko Endeavouria mahdollista käyttää CP-E2:en järjestelmäkomponenttitestauksessa yrittämällä toteuttaa siihen tarvittavat toiminnot.

CP-E2:n testaukseen tarvittavien toimintojen lisääminen Endeavouriin vaati lähinnä SCTP-tuen toteuttamisen Endeavourin ja CP-E2:n välillä. Tätä varten oli tehtävä kaksi uutta luokkaa: SCTP-rajapinta ja SCTP-serialisaattori. Rajapinta luo SCTP-yhteyden ja käsittelee sen kautta lähetettävät viestit. Serialisaattoria käytetään testaussanomien serialisointiin Endeavourissa käytettävään muotoon.

Endeavouria kehitettiin, kunnes kaikki paitsi muutama CP-E2:en testiä oli testattavissa Endeavourin avulla. Koska syy näiden muutaman testin epäonnistumiseen oli tiedossa, pystyttiin osoittamaan Endeavourin soveltuvuus CP-E2-komponentin testaamiseen. Tehdyn työn perusteella voidaan todeta, että Endeavouria pystyy käyttäämään CP-E2:en testaamisessa.

Avainsanat Ohjelmistokehitys, SCTP, C++, 5G
Sivut 35 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and communications technology Abstract
Author Leevi Räsänen Year 2022
Subject Development of Nokia's 5G testing tool: Expanding functionality to CP-E2 component
Supervisor Toni Laitinen

---

Nokia is investigating if a tool called Endeavour can be used to improve Nokia's 5G software testing procedure in the 5G control plane. Presently, the main goal of Endeavour's development is to see if it can be a viable testing tool. Part of this viability evaluation is implementing support for additional control plane components to be testable in Endeavour. There is an interest in seeing if one such component, CP-E2, can be supported. In this thesis, it is explored if it is feasible to use Endeavour in CP-E2 system component tests by trying to implement the necessary functionality for doing so.

Adding the required functionality for CP-E2 testing into Endeavour mostly centered around implementing support for SCTP between Endeavour and CP-E2. For this, two new classes had to be made: SCTP interface and SCTP serializer. The interface sets up the SCTP connection and handles the messages sent over it. The serializer is used to serialize testing messages into a format that is usable in Endeavour.

In the end, Endeavour could be used to correctly execute all but a tiny minority of existing CP-E2 tests. And, as the cause of the last few tests' failure was known, more than enough had been done to show how viable Endeavour is for CP-E2 testing. Based on the work done it can be said that Endeavour can be used in place of TTCN-3 in CP-E2 testing, but whether it actually will be is yet to be determined.

Keywords Software development, SCTP, C++, 5G
Pages 35 pages

## Table of contents

## List of abbreviations

| | |
|---|---|
| 3GPP | 3rd Generation Partnership Project |
| 5G | fifth-generation |
| ASN.1 | Abstract Syntax Notation number One |
| CP | control plane |
| CP-E2 | control plane E2 |
| CU | central unit |
| CU-C | central unit control plane |
| CU-U | central unit user plane |
| DU | distributed unit |
| GUI | graphical user interface |
| JSON | JavaScript Object Notation |
| MCT | multi component test |
| PDU | protocol data unit |
| PER | Packed Encoding Rules |
| RAN | radio access network |
| RU | radio unit |
| RRC | radio resource control |
| SCT | system component test |
| SCTP | Stream Control Transmission Protocol |
| SUT | system under test |
| TTCN-3 | Testing and Test Control Notation version 3 |
| UE | user equipment |
| UT | unit test |
| UP | user plane |
| XML | Extensible Markup Language |

# 1 Introduction

Nokia is one of the key companies in the telecommunications industry. A lot of new telecommunications infrastructure, including Nokia's, is focused on utilizing 5G, the newest in-use mobile network. Due to this, Nokia is trying to improve its software development process for 5G if something could be sped up or otherwise refined (Nokia, 2021). One such instance was noticed in the 5G control plane's system component testing procedure and as a result the development of a testing tool, Endeavour, was started.

Endeavour is still in development but it can already be used to run system component tests for some 5G control plane components. The main goal of Endeavour's development at the moment is to see if it can be a viable testing tool. Part of this viability evaluation is implementing support for additional control plane components to be testable in Endeavour. There is an interest in seeing if one such component, CP-E2, can be supported. In this thesis, it is explored if it is feasible to use Endeavour in CP-E2 system component tests by trying to implement the necessary functionality for doing so.

# 2 Background information

There is a lot of information about 5G, software development and specific programming languages that needs to be understood before moving into the implementation. While a lot of the theoretical background was not needed for the practical implementation, it helps to understand the aims and purpose of Endeavour and this work, and the reason behind some of the decisions made during the development.

## 2.1 5G

Fifth-generation (5G), as in the fifth-generation of mobile networks, is the newest commercially available generation of mobile networks. 5G networks have been in use since 2019 and the amount of them has been constantly increasing since then. There is no single

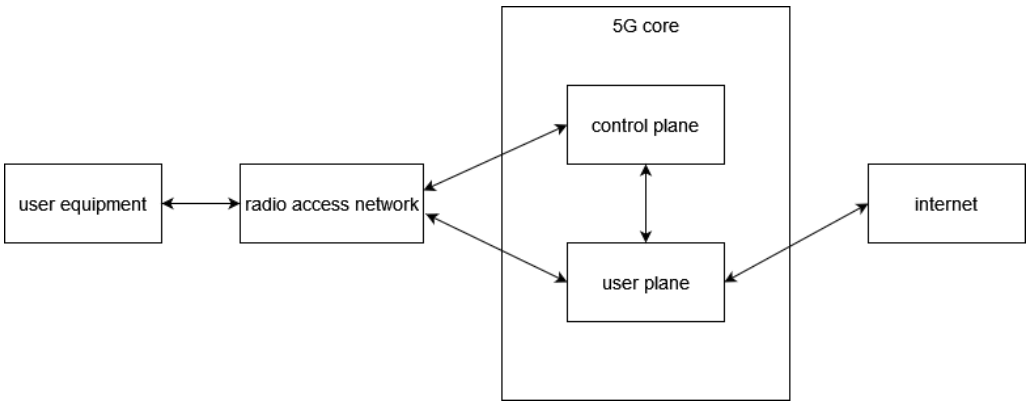
inventor or owner of 5G, instead the 3rd Generation Partnership Project (3GPP) defines the technical specifications of 5G. (Qualcomm, 2022)

3GPP unites multiple telecommunications development organizations to help them produce material that is then used to define 3GPP technologies, including 5G. 3GPP was formed in 1998 just for 3G, but the project has since expanded in scope (3GPP, 2015). 3GPP does not just monolithically define how 5G works, as companies like Nokia have representatives in the project (Israel & Gardella, 2021).

5G consists of two different large-scale components: the radio access network (RAN) and the 5G core. RAN handles the connections from and to user equipment (UE) and manages these connections. UE is any sort of end device that is used to connect to the 5G network, for example, a smartphone, a home appliance, or even a car. The core handles multiple different tasks such as providing internet connectivity to UEs and tracking usage for user billing purposes. (Peterson & Sunay, 2020)

The 5G core can be split into two parts, the control plane (CP) and the user plane (UP), both of which can be further broken down into smaller microservice-like software components. In the UP there is a single component, the user plane function, whose main purpose is forwarding traffic between the RAN and the internet (Peterson & Sunay, 2020). Figure 1 highlights the important parts of high-level 5G architecture.

Figure 1: High-level 5G architecture.

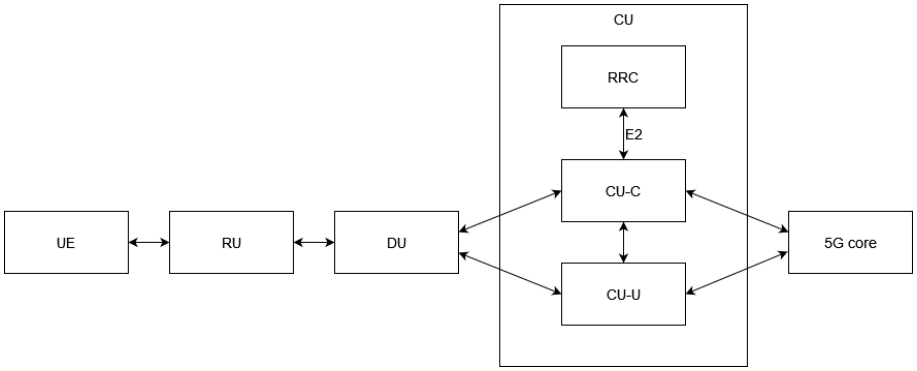



Control plane is not an exclusive concept to 5G core, and in networking can mean any layer of a network that manages how data moves through that network (Cloudflare, 2020). In the 5G core, the CP is responsible for exactly this purpose: it ensures that user equipment can efficiently access services, i.e. data in the form of voice or text, in data networks (Ali & Guttman, 2018).

The RAN in 5G is flexible and can be configured in many different ways, of which a three-part split is a common option. In this configuration RAN consists of three parts: the radio unit (RU), distributed unit (DU) and central unit (CU). UEs connect to a RU which converts radio signals to digital signals for further transmission into a DU. The DU then connects to a CU, which then finally connects to the 5G core. (Peterson & Sunay, 2020)

The CU can be split into a control plane and a user plane, similarly to the core. These parts are known as the central unit control plane (CU-C) and the central unit user plane (CU-U). Furthermore, in accordance with software-defined networking principles the component known as radio resource control (RRC) can be separated from the rest of the CU-C. RRC connects to the CU-C through the E2 interface (Peterson & Sunay, 2020). The relevant architecture of 5G RAN can be seen in figure 2.

Figure 2: 5G RAN architecture.




Control plane E2 (CP-E2) is a component of the 5G network that exists inside CU-C for the purpose of managing the E2 interface between CU-C and RRC. CP-E2 is the component for which Endeavour's testing capabilities are expanded, but despite this, its precise purpose and methods are largely inconsequential.

## 2.2 Software testing

Software testing is the process of checking if software functions as it is actually expected to. Testing is an incredibly important aspect of software development as it improves reliability and typically saves time in the long term. (Hamilton, *What is Software Testing? Definition, Basics & Types in Software Engineering*, 2022)

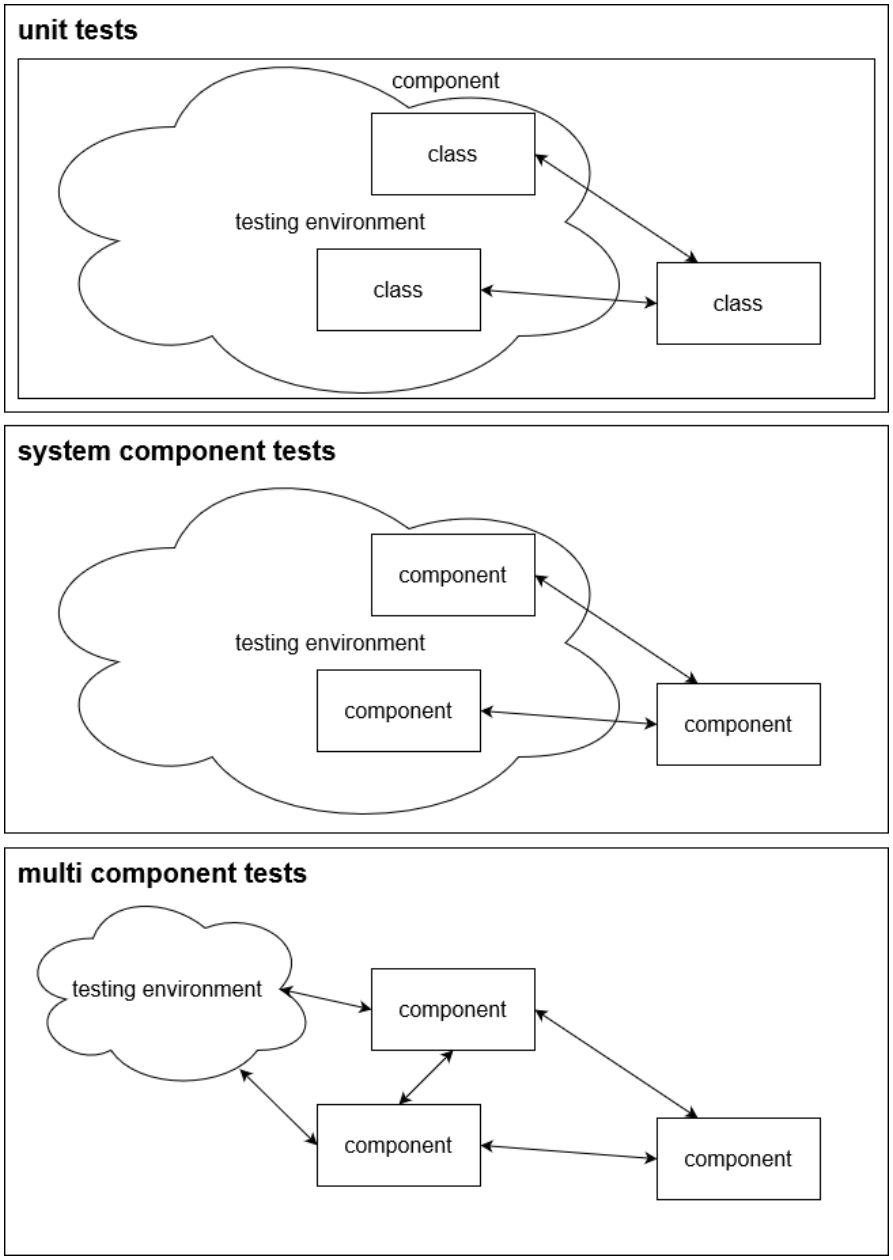
Testing is at least tangentially relevant to most types of software development projects and very important for Endeavour, as not only does Endeavour have to be tested itself, it also needs to test other software. There is a large amount of different software test types, but in the scope of Endeavour, only two are particularly relevant. Unit tests (UTs), which test the functionality of a small part of the code e.g. one function or class (Fowler, 2014), are important as they are used to test Endeavour's internal robustness. The second type is component tests which are relevant as Endeavour is built to run component tests for 5G software. Component tests are not as straightforward as unit tests and are more significant for Endeavour so they need to be examined in more detail.

Component testing is the process of testing some single complete component of a system. If the component does not need to interact with other components, then the testing can be relatively straightforward. However, if the component needs to interact with others, then either the other components need to be loaded in addition to the one under test or the other components can be substituted with fake components of some kind. (Hamilton, *What is Component Testing? Techniques, Example Test Cases*, 2022)

Component testing is the main purpose of Endeavour so it needs to be understood quite well, unlike unit testing which is only needed for internal correctness and stability. Not to understate unit testing's importance, as it is still very important. The difference between the two testing types might seem somewhat arbitrary as neither unit nor component is a clearly defined concept. The difference is easily understood when looking at how each testing type is done in practice. Unit testing is what a developer does to test if what they wrote works as expected. On the other hand, component testing is done by a tester after the code has been written to test if the software works correctly in a real use-case scenario. (Hamilton, *What is Component Testing? Techniques, Example Test Cases*, 2022)

The component tests in 5G CP almost always test a single component at a time with test doubles being used in place of other components necessary for the tests. At Nokia, these types of tests are referred to as system component tests (SCTs) to distinguish them from multi component tests (MCTs) which test multiple real components simultaneously. The distinction also helps to express that in SCTs only a single component is being tested at a time. The component that is being tested is referred to as a system under test (SUT). The SUT in Endeavour and TTCN-3 testing for the control plane is a component of the control plane and, for the purposes of this work, the SUT in SCTs is specifically the CP-E2 component. The difference between UTs, SCTs and MCTs is further expressed in figure 3.

Figure 3: Different testing methods visualized.




## 2.3 TTCN-3

Testing and Test Control Notation version 3 (TTCN-3) is a test specification language maintained by the European Telecommunications Standards Institute (TTCN-3.org Editorial Team, 2013). TTCN-3 is the traditional way of writing SCTs for 5G. TTCN-3's use as the standard language for 5G testing came about naturally as TTCN-3 was already endorsed by

3GPP for 4G testing and was planned to be used for all future test development (Willcock, et al., 2011).

TTCN-3 only needs to be understood to the point where some of the advantages and disadvantages it has in testing can be recognized. This is because Endeavour tries to provide improvements in some of these areas compared to TTCN-3 testing, so only a baseline is needed to see if and how Endeavour might improve upon it. Testing with TTCN-3 is not a complicated process: the tester needs to write a test in TTCN-3, after which the test can be run. However, one significant drawback is that TTCN-3 is quite a niche language and as such most new employees need to learn the language from scratch. Despite that, for an experienced user the language is fundamentally suitable for writing tests.

## 2.4 Endeavour

Endeavour is Nokia's testing tool for the 5G control plane. Endeavour was originally conceived during 3G development where it was used as a testing tool. The initiative to start developing a version of Endeavour for 5G development came as a result of an attempt to see if SCT test development could be accelerated as it was perceived to be a time-consuming part of development. Endeavour for 5G is still in a proof of concept phase. This itself has many effects on how the development is handled as it means that rapid development is prioritized. This in turn means that things like software robustness, while not ignored, take slightly less precedence. As a result, usually favored development techniques, such as test-driven development, are not always complied with. In the same vein, high test coverage is not expected for Endeavour as it can be implemented later on if the project is seen valuable enough for further development. Currently, Endeavour is being developed to the point where its viability for testing 5G CP components can be properly evaluated.

Most of the 5G CP SCTs test that the messages between the SUT and other components arrive and leave in an expected order. For the SUT to perform some desired action, specific messages have to be sent to it. For example, if some service is wanted, a request for it has to be sent. While these types of SCTs can be done in TTCN-3, the visual representation that Endeavour offers expresses explicitly that the SCTs are about the messages and their order.

Instead of writing code directly, messages are created with the expected contents, the expected order and the expected sender and receiver. As the test case then consists of only messages, there is no room for confusion. In both TTCN-3 and Endeavour, the singular messages can be hidden away under layers of abstraction to simplify the tests.

One advantage that Endeavour offers is that there is no need for a tester to learn the TTCN-3 language. As a test is made with Endeavour's simple-to-use graphical user interface (GUI) there is no need to learn an entirely new language and its syntax to be able to make SCTs. Another advantage is that Endeavour clearly and explicitly shows that the 5G CP SCTs are about the order of messages and the correctness of their contents. If a test fails due to some unexpected message contents, the tester can easily find the message in the test case and compare the correct and unexpected content.

While Endeavour's architecture is quite complicated, it can be split into three different components to make it easier to understand. These components are the user interface, the Endeavour server, and the Endeavour converter. The GUI is made in C# and the server and converter are written in C++. Expanding Endeavour's functionality to CP-E2 only requires changes to the server and the converter.

Endeavour tests, which ultimately are Extensible Markup Language (XML) files, can be made in or exported to Endeavour's GUI. The test execution is started from the GUI but requires the Endeavour server to be running. The main testing logic of Endeavour is implemented in the server, which, if run by itself, does nothing but wait for a test to be started from the interface. Once a test is running the interface and server share information to run the test and show the message flow in real time. While the interface and server require each other the converter is more of a stand-alone tool. The purpose of the converter is to convert existing TTCN-3 SCTs to Endeavour tests. Without it, already written SCTs would have to be manually reimplemented in Endeavour. An example of a test's underlying XML file can be seen in figure 4.

Figure 4: Two messages from an XML file that makes up an Endeavour test.

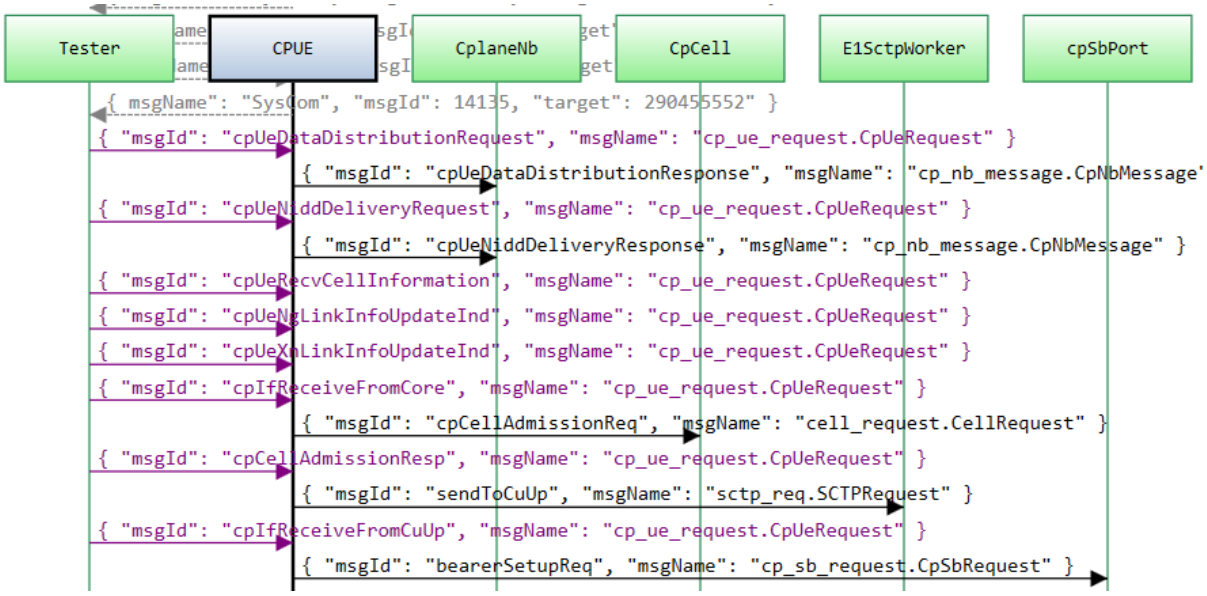
```
<HttpMessage>
    <ToBts>@2</ToBts>
    <FromBts>@1</FromBts>
    <Header>{ "msgId": "cpE2DataDistributionRequest", "msgName": "cp_e2_interface.CpE2Request" }</Header>
    <Body>{"cpE2DataDistributionRequest":{"transactionId":1,"cpnbPoolId":60000,"cpnbNodeAddress":4449,"cpnb
    <JsonMessageScripts />
    <Enabled>0x1</Enabled>
    <MessageInfo>
        <MessageNumber>0x0</MessageNumber>
    </MessageInfo>
    <ForceToSimulatedMessage>0x0</ForceToSimulatedMessage>
    <ForceToMonitoredMessage>0x0</ForceToMonitoredMessage>
    <DontVerifyMessageData>0x0</DontVerifyMessageData>
    <Attribute>0x1</Attribute>
    <SendToPhysicalAddress>0x0</SendToPhysicalAddress>
</HttpMessage>
<HttpMessage>
    <ToBts>@5</ToBts>
    <FromBts>@2</FromBts>
    <Header>{ "msgId": "cpE2DataDistributionResponse", "msgName": "cp_nb_message.CpNbMessage" }</Header>
    <Body>{"cpE2DataDistributionResponse":{"transactionId":1,"status":"ok"}}</Body>
    <JsonMessageScripts />
    <Enabled>0x1</Enabled>
    <MessageInfo>
        <MessageNumber>0x0</MessageNumber>
    </MessageInfo>
    <ForceToSimulatedMessage>0x0</ForceToSimulatedMessage>
    <ForceToMonitoredMessage>0x0</ForceToMonitoredMessage>
    <DontVerifyMessageData>0x0</DontVerifyMessageData>
    <Attribute>0x1</Attribute>
    <SendToPhysicalAddress>0x0</SendToPhysicalAddress>
</HttpMessage>
```

Endeavour converter hooks a TTCN-3 test executable to save the messages transferred during the test's execution to an XML file. In a hooked test, some functions are hooked to make test conversion possible. These functions are mainly the ones responsible for sending and receiving messages through different transport protocols. The hooked functions save the messages they handle in a serialized format to the converted test in addition to doing their normal non-hooked tasks. A test ran like this still gives a verdict based on whether it fails or passes. In a hooked test's case, failure means that the test conversion was not successful. On the other hand, a pass does not guarantee that the converted test is going to be functional, instead, it simply confirms that the hooked TTCN-3 test passed and that the converter did not run into errors. During the conversion, only some of the expected messages are compared against the ones that are actually transmitted, therefore some missing messages are possible. Additionally, if the test uses, for example, a transport protocol, it would have to be set up separately in the Endeavour server while during conversion the TTCN-3 test sets it up. Both of these reasons can result in a converted test's failure when ran with Endeavour.

Endeavour tests consist of messages which have a sending and a receiving component. These components are components of the 5G CP or a tester. One of the 5G CP components serves as the SUT while the others are test doubles. The tester component is implemented in the Endeavour server. It sends messages to the SUT, which results in the SUT performing one or more desired actions. For example, the tester might send a request to the SUT which then sends a response to the tester indicating if it could accommodate the request. The messages also have a header and body, both of which are in JavaScript Object Notation (JSON). The body is a serialized form of the actual content of the message, while the header consists of information such as the message's name, type, or the protocol through which it is sent.

Figure 5 shows how a test is displayed in Endeavour's GUI. The components needed for the test are depicted by the green and blue rectangles at the top of the view. The tester is the leftmost component, CP-UE to its right serves as the SUT for this test and the rest of the components are test doubles. Each arrow represents a message with the arrow beginning from the sender and pointing to the receiving component. Above each message is its header. The contents of the messages are not visible in the figure, but they can be inspected in the GUI.

Figure 5: A test in Endeavour's GUI.




When a test is run, the tester, the component serving as SUT, and fakes of the other components are set up and then the messages are iterated through. For each message by the tester, the header is used to determine how the message is sent to the SUT and the body of the message is deserialized into sendable data. When the SUT receives a message from the tester it might send additional messages to the tester or the other components. As these messages are transferred between the components, they are also sent to the Endeavour server to be compared against the messages in the test case. A discrepancy between a message that is sent during execution and a message in the test case results in the test failing. Out-of-order messages and missing messages both result in failure as well. If all of the messages in a test case appear correctly during a test's execution, the test passes.

## 2.5 C++

C++ is a general-purpose programming language invented by Bjarne Stroustrup in 1979 and was commercially released in 1985 (Standard C++ Foundation, 2014). C++ is used in 5G software development by Nokia. Endeavour is mostly written in C++, so proficiency in it is necessary for development. While C++ knowledge was absolutely necessary for the development, it is not as necessary for understanding this thesis as the code implementation is not always explored on a line-by-line basis. Some specific functions of C++ and some C++

libraries are mentioned directly if they are important enough to warrant it. The specific version of C++ that is used in Endeavour is C++17, therefore many features of modern C++ can be taken advantage of.

C++ is used in Endeavour mainly because it is already used in 5G development at Nokia, so the developers working on Endeavour do not have to learn a new language just for developing Endeavour. Performance also plays a role as SCTs are rarely executed one at a time and a more common use case is running all of the tests for a component or multiple components. This means that often hundreds, if not thousands, of tests have to be executed back to back, which causes even small delays to add up. Endeavour's viability as an alternative to TTCN-3 would be reduced if the test execution time would increase significantly.

## 2.6 SCTP

Stream Control Transmission Protocol (SCTP) is an IP transport protocol first specified in RFC 2960 in 2000 and introduced in RFC 3286 in 2002 (Ong & Yoakum, 2002). The RFCs are technical specifications produced by the Internet Engineering Task Force which specify protocols such as Transport Layer Security 1.3 and Web Real-Time Communication in addition to SCTP (IETF, 2017).

SCTP is relevant for this work, as CP-E2 uses it to communicate with other 5G components. There was no support for SCTP in Endeavour so it had to be implemented. For this project, the most important part of SCTP is its one-to-one messaging process, which is explored in detail during the implementation when it becomes relevant to the development.

## 2.7 ASN.1

Abstract Syntax Notation number One (ASN.1) is a notation standardized by the International Telecommunication Union Telecommunication Standardization Sector used to describe data (ITU, 2020). ASN.1 has been in use since 1984 and is particularly relevant in the field of telecommunications (ITU, 2020). ASN.1 is used to encode information so that it can

be transferred while explicitly instructing how any part of the information should be represented, which allows the information to be independent of programming language, hardware and operating system (OSS Nokalva, 2018).

ASN.1 defines many encoding rules of which Packed Encoding Rules (PER) are of particular relevance as they are used for encoding in CP-E2. Data encoded with PER is very compact as it does not send information already known in the data's ASN.1 schema in order to minimize the use of bandwidth (OSS Nokalva, 2006). This compactness comes at a cost, as the schema has to be known by the decoder, or else the process of decoding is difficult or, at worst, impossible. As with SCTP, there was no handling for ASN.1 in Endeavour present and thus it had to be added.

# 3 Purpose and aims

The aim of this work is to implement the required functionality for Endeavour to be usable in CP-E2 testing for the purpose of allowing Endeavour's viability in CP-E2 testing to be examined. If Endeavour is going to be used in the 5G SCT process for CP-E2, then the work done allows the benefits of Endeavour to be realized in a new area. This in turn would increase Endeavour's intrinsic value as it would gain more use cases.

The exact point where it can be stated that enough is done for Endeavour's viability in CP-E2 testing to be observed is not obvious. While being able to convert and execute every existing CP-E2 SCT with Endeavour would certainly meet the goal, it is not exactly necessary. The implementation can be done to the point where it can be reasonably determined whether Endeavour is fit for the task. So it is enough that only some of the SCTs are usable in Endeavour, as long as it can be determined whether the rest can also be used, without necessarily implementing the functionality to do so.

To meet the aim, new functionality has to be added to Endeavour in the form of additional code. In addition, it is very likely that the existing code has to be edited. To be able to accomplish the aim Endeavour's existing implementation has to be analyzed in great detail before any new additions can reasonably be made.

# 4 Planning and implementation

Adding the ability to test CP-E2 with Endeavour was done as a part of a normal software development cycle. There are multiple developers working on Endeavour simultaneously and the work is coordinated in daily meetings. Before any new code could be written, a development environment had to be set up and the existing code of Endeavour had to be analyzed to get an idea of where and what had to be added. Only then could the new functionality be actually implemented. Once the necessary changes and additions were made, the new code was added on top of Endeavour, after which the end result was scrutinized to see if and how well the aims of this work were reached.

## 4.1 Development environment

The development was done on a virtual machine running Community Enterprise Operating System Linux distribution as its operating system. Secure Shell Protocol was used to connect to the virtual machine. The source code of Endeavour is remotely stored in a repository that is hosted on GitLab and Git is used for version control and interacting with the remote repository. Microsoft's Visual Studio Code was used to edit the code. The first step in starting to work with Endeavour was cloning the repository to the virtual machine, after which it was possible to build Endeavour and use the Endeavour converter and server. It would have been possible to start editing the code to extend Endeavour's functionality to CP-E2 at this point, but instead, the code was first analyzed to gain a better understanding beforehand.

## 4.2 Analyzing Endeavour

The logical starting point for adding new functionality to an already existing project is studying the structure of the current implementation (McGregor, 2020). Endeavour's implementation was only analyzed until a basic understanding was reached. While it would have been possible to try to understand every single aspect in detail, it likely would not have been worth the effort as there are aspects of Endeavour that were not necessary for this

work. Instead, during the development additional parts were explored more in-depth whenever it was necessary.

During the analysis, one peculiar aspect of Endeavour was revealed: Endeavour converter and server are quite intertwined in the code. In the CMakeList that compiles Endeavour two different libraries are made: Endeavour converter and Endeavour server. For the server library, a preprocessor variable indicating that it is the server is defined with target_compile_definitions. In the code, preprocessor directives are used to conditionally include parts of the code that are necessary for only one of the modes based on whether the preprocessor variable is defined in the library that is being compiled. The reason for the two different modes is mainly efficiency. Most of the functionality in Endeavour is shared by both the converter and server so writing two separate programs would mean that a lot of the code would be represented in two places at once. This would mean that any change to the shared portions would have to be made to both of the representations at once. This would be quite cumbersome and slow the development down as well as increase the chance of making mistakes.

### 4.3 Enabling the CP-E2 component to be used in Endeavour testing

The first problem to solve was getting Endeavour to work with the CP-E2 component. Initially, Endeavour was made to work with a different component, CP-UE, with no immediate support for others. However, a lot of the currently implemented functionality would work with CP-E2 as well, so only certain hard-coded values needed to be changed initially.

For the Endeavour converter to work a TTCN-3 SCT needs to be executed, this creates an executable file. Endeavour needs this executable to convert a TTCN-3 test to XML. Endeavour converter is ran by running a shell script which later into the execution also calls a separate script which hooks the executable. Both of these scripts have multiple references to CP-UE-specific values. Simply changing the CP-UE values to CP-E2 equivalents is enough to fix them and allows Endeavour to be ran with CP-E2 tests instead of CP-UE ones.

Now that Endeavour could at least compile with the new component, a CP-E2 test needs to be built to continue. For this purpose, a simple test is optimal as it is potentially easier to first implement the bare minimum needed for a test to work with Endeavour, and then later check with larger and more complicated tests to see if additional changes are needed. There is one SCT for testing if CP-E2 can start successfully, which was chosen as it appeared to be the simplest CP-E2 test. The test just had to be ran normally in order to generate the executable.

With the changes made and the executable in place, the Endeavour converter could be built. Running the test with the hooked executable resulted in an unsurprising failure. Endeavour already had vast logging capabilities which made solving the failure easy. Most 5G components in CP need to establish communications with other components in some way. Endeavour already has support for ZeroMQ (ZMQ) and logs if there are unimplemented ZMQ ports. These logs reveal that there was a missing CP-E2 ZMQ port. This port is what CP-E2 uses to communicate with some other CP components so it needed to be implemented.

Adding the CP-E2 port was straightforward as everything required for the whole ZMQ process had already been implemented in Endeavour. Endeavour sets up all the needed ZMQ ports in a specific class which is called at the start of Endeavours execution. Thus the CP-E2 ZMQ port could just be created in that same class. As CP-UE also has a ZMQ port that is implemented there, it was possible to just match what was done to implement it and do the same for CP-E2's port. Running the test through the Endeavour converter again with the port added resulted in the test passing successfully.

### 4.4 Hooking SCTP functions

With the ZMQ port working, the next logical step was trying to run the test in the Endeavour server to see if it also succeeds. The test ran smoothly up until it ran into a problem: one message the test was expecting never arrived causing the test execution to be stopped. Looking at a separately made diagram of the expected message flow revealed that there were three messages that were supposed to transfer before the missing one, but they were not found in the converted test. The first problem to solve was the missing messages, as

their absence was a probable cause for the one message to never be transferred. From the diagram, it could be deduced that at least the first of the three messages was one sent over SCTP based on the message's name, sctp_assoc_change. Endeavour did not yet have support for SCTP, and therefore the three messages were not converted from the TTCN-3 test. This is why the Endeavour test did not fail due to the three missing messages as Endeavour did not have knowledge of them even being supposed to be there.

To proceed support for SCTP-based messages had to be added. This was accomplished by hooking the original SCTP symbols which the TTCN-3 tests use, in order to show what messages are being transferred through them in Endeavour. Running the initial TTCN-3 test builds a shared object library for SCTP which holds the original symbols. Endeavour converter is injected using LD_PRELOAD Unix environmental variable which enables the ability to hook these symbols by creating new identical ones. In the function body of these new symbols, the desired functionality can be implemented. The use of LD_PRELOAD causes the new symbols to be loaded before the original ones and as such, they are called instead. To create the overwriting symbols, functions with identical names, return types and arguments had to be made inside an extern "C" block to avoid C++'s name mangling. This way of hooking was chosen as it was already used in Endeavour to implement, for example, the earlier mentioned ZMQ support.

In order to avoid having to rewrite the functionality of the original SCTP functions a method of accessing the original symbols was implemented. This was done by creating new function pointers pointing to the original symbols with <dlfcn.h>'s dlsym. These new functions could then be used to call the original functions even after the hooking. The process of creating both a new symbol and a new function for calling the original one was done for all relevant SCTP function symbols in the shared object library.

## 4.5 SCTP interface

Next, a class for the SCTP interface was created and in this class, an implementation for all the newly defined overwrite functions was made. To see which SCTP functions are truly relevant each of the implementations was made to just print the function's name to stderr,

i.e. the standard error output stream, and return the value obtained from a call to the original symbol. Then the test was converted again to see which symbols ended up being called. From Endeavour's output, it was perceptible that the functions that are called are sctp_bindx, sctp_connectx, sctp_recvmsg, sctp_sendmsg and then sctp_recvmsg again. It was certain that recvmsg and sendmsg would need some handling to have the messages show up in Endeavour, but bindx and connectx could be left as is.

To figure out how sctp_recvmsg should be handled, Nokia's 5G codebase was searched for its usage to see how it was being used in a 5G context. There was one very useful example which was used as a reference in addition to public SCTP information. Importantly in the example <netinet/sctp.h> was included, which has many SCTP-related definitions and therefore was also included in the SCTP interface.

One parameter of sctp_recvmsg is msg_flags, which was used in the example to check whether the message received was an SCTP notification or some other message. SCTP notifications are messages containing information about events and errors on the SCTP stack (Stewart, Tuexen, Poon, Lei, & Yasevich, 2011). On the other hand, non-notification messages are the actual messages being transmitted between CP-E2 and other components. Adding separate print statements for both outcomes revealed that the first recvmsg call was for a notification while the second was for a non-notification. The example where sctp_recvmsg was used included lots of useful information including how to get the type of the notification. Implementing similar functionality in the SCTP interface revealed that the notification was of type sctp_assoc_change. Figure 6 shows how sctp_assoc_change is defined in <netinet/sctp.h>.

Figure 6: sctp_assoc_change's definition in <netinet/sctp.h>.

```
/*
 * 5.3.1.1 SCTP_ASSOC_CHANGE
 *
 *   Communication notifications inform the ULP that an SCTP association
 *   has either begun or ended. The identifier for a new association is
 *   provided by this notificaion. The notification information has the
 *   following format:
 *
 */
struct sctp_assoc_change {
        __u16 sac_type;
        __u16 sac_flags;
        __u32 sac_length;
        __u16 sac_state;
        __u16 sac_error;
        __u16 sac_outbound_streams;
        __u16 sac_inbound_streams;
        sctp_assoc_t sac_assoc_id;
        __u8 sac_info[0];
};
```

In order to show the contents of the SCTP association change in Endeavour, it needs to be converted from a void pointer, in which it is received in sctp_recvmsg, into a JSON style string. The first step is copying the notification from the void pointer to a type. The desired type is sctp_assoc_change as defined by <netinet/sctp.h>, but copying the data to it directly would potentially be unsafe and cause problems if other SCTP notifications would have to be supported. So a more generic solution was needed. <netinet/sctp.h>'s sctp_notification is a tagged union that holds sn_header, which is a struct containing the tag, i.e. the type of notification it contains, flags and the length of the notification. All of the notifications also have these same fields as their first bytes. Therefore a sn_header can be created and then memcpy can be used to copy as many bytes to it as the header can hold. From this header the notification's type, flags and length can be accessed. The flags are inconsequential for now, but with the type and length, the data can be copied safely. A switch statement was made with the type as its expression. In this statement, a case for sctp_assoc_change was declared in which memcpy is used to copy as many bytes from the void pointer to it as were defined in the length field. The switch statement's default case was made to print the type to stderr so if there was a need to implement serialization for other notification types it

could be easily seen. Sctp_notification's definition in <netinet/sctp.h> is shown in figure 7 and the code written for accessing the type can be seen in figure 8.

Figure 7: sctp_notification's definition in <netinet/sctp.h>.

```
/*
 * 5.3.1 SCTP Notification Structure
 *
 *   The notification structure is defined as the union of all
 *   notification types.
 *
 */
union sctp_notification {
        struct {
                __u16 sn_type;             /* Notification type. */
                __u16 sn_flags;
                __u32 sn_length;
        } sn_header;
        struct sctp_assoc_change sn_assoc_change;
        struct sctp_paddr_change sn_paddr_change;
        struct sctp_remote_error sn_remote_error;
        struct sctp_send_failed sn_send_failed;
        struct sctp_shutdown_event sn_shutdown_event;
        struct sctp_adaptation_event sn_adaptation_event;
        struct sctp_pdapi_event sn_pdapi_event;
        struct sctp_authkey_event sn_authkey_event;
        struct sctp_sender_dry_event sn_sender_dry_event;
};
```

Figure 8: The code for getting an SCTP notification's type.

```
std::uint16_t getNotificationType(const void* msg, const size_t msgSize) {
    constexpr size_t headerSize{ sizeof(sctp_notification::sn_header) };
    if (msgSize < headerSize) throw std::runtime_error{ "SCTP notification is smaller than the notification header" };

    sctp_notification notification{};
    auto& header = notification.sn_header;
    std::memcpy(&header, msg, headerSize);

    if (msgSize != header.sn_length) {
        throw std::runtime_error{ "SCTP notification size does not match reveived notification's size" };
    }
    return header.sn_type;
}
```

The rest of the serialization is a simple process of creating a JSON key for association change and setting its value to an array of JSON key-value pairs based on each variable in the received sctp_assoc_change and their respective values. This was done with Subhrajit Bhattacharya's RSJp-cpp tool, which makes it possible to both compose strings into JSON and

parse JSON into strings. It would have been possible to do the same with raw strings, but that would have been a less readable, harder to change and more error-prone alternative. The JSON made with RSJp-cpp is then composed into a string, which is the final serialized form of the notification. The whole process of converting an association change from a void pointer to a string was moved into a separate serialize function in a new SCTP serializer class. The code written for composing sctp_assoc_change into JSON can be seen in figure 9. The extraBytes parameter and the line where the key-value pair for sac_info is made were added later on in the development. Figure 10 shows how the serialized message appears in Endeavour.

Figure 9: The code for composing sctp_assoc_change's fields into JSON with RSJp-cpp.

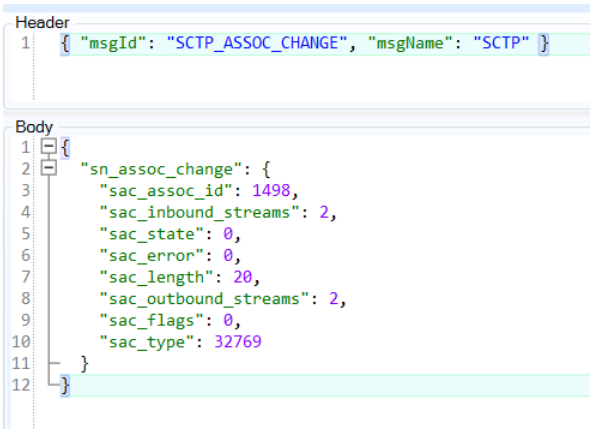
```
RSJresource serializeAssocChange(const sctp_assoc_change& sac, const int extraBytes) {
    RSJresource jsonNotification{ "{ \"sn_assoc_change\": {} }" };
    RSJresource& jsonSac = jsonNotification["sn_assoc_change"];

    jsonSac["sac_type"]             = sac.sac_type;
    jsonSac["sac_flags"]            = sac.sac_flags;
    jsonSac["sac_length"]           = sac.sac_length;
    jsonSac["sac_state"]            = sac.sac_state;
    jsonSac["sac_error"]            = sac.sac_error;
    jsonSac["sac_outbound_streams"] = sac.sac_outbound_streams;
    jsonSac["sac_inbound_streams"]  = sac.sac_inbound_streams;
    jsonSac["sac_assoc_id"]         = sac.sac_assoc_id;

    if (extraBytes) jsonSac["sac_info"] = serializeSacInfoAbortChunk(&sac.sac_info[0], extraBytes);

    return jsonNotification;
}
```

Figure 10: Serialized SCTP association change in Endeavour's interface.

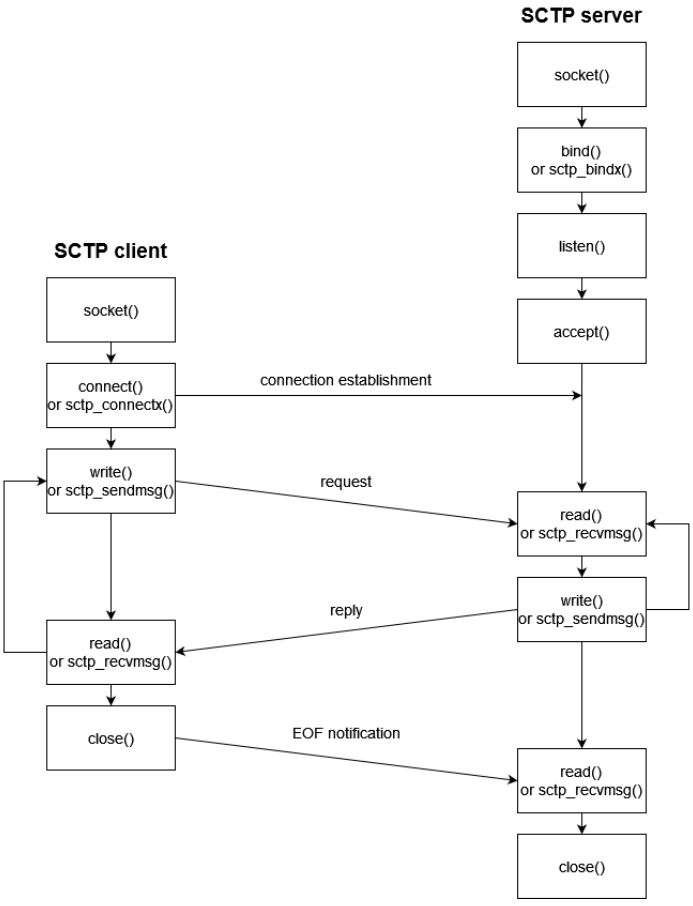



A normal received SCTP message would be added to a converted test in the interface's sctp_recvmsg. However, as SCTP association change is a notification it should not be included in converted tests. The notifications are not something that should be tested in Endeavour as their values are constantly changing. For example, association change holds the id of the SCTP association. This id can vary even in otherwise identical tests. So a converted association change's id would most likely be different than the one send during an actual test execution resulting in a discrepancy and thus the test's failure. Instead, serialization is needed to show that it is being sent in the Endeavour server. But before it can be used there the interface had to be changed to work with the Endeavour server. A similar interface for the server could be created but a simpler solution is using the same interface for both modes of Endeavour with the help of the aforementioned preprocessor variable. The #ifdef directive can be used to determine if the variable is defined and thus whether it is the server or the converter that is being used. In the interface's sctp_recvmsg it is determined if the incoming message is a notification. Inside an "if statement" with the condition of the message being a notification, an #ifdef block with the preprocessor variable as an identifier was added. In this block, the received notification message is serialized with the new serializer and then send to Endeavour server monitoring with the help of another Endeavour component. Because notifications are not added to Endeavour tests they have to be sent to the server as messages that are not compared against the messages in the test case. If they were to be added normally the test would fail after a notification is sent to the monitoring as there would not be a matching message in the test.

## 4.6 Establishing an SCTP server

With the first sctp_recvmsg call diagnosed as an sctp association change and handled the next step was figuring out the sctp_sendmsg call and the other sctp_recvmsg call. Running the converted test with the Endeavour server resulted in a failure in the middle of the test's execution. The logs revealed that there was an SCTP client socket that failed to connect to a server causing the test to fail. This was not a problem with the Endeavour converter as there the TTCN-3 test sets up the SCTP socket, but in the server, it was not yet implemented. The socket is created in order to establish a one-to-one style SCTP interface between the tester and the CP-E2 component. Figure 11 shows an example timeline of the system call sequences of a one-to-one style SCTP interface and a client.

Figure 11: SCTP one-to-one system call sequences (Stevens, Fenner, & Rudoff, 2003).




Getting the socket ready to listen is quite a simple task. First, the socket can be created with <sys/socket.h>'s socket which returns a new file descriptor. This socket is the one that is

going to serve as the server socket. Then an address has to be bound to this socket with the original sctp_bindx so that the client can connect to it. It does not matter what the address is, as long as the client knows it. Before an SCTP connection is established between the tester and the CP-E2 component a message is sent between them to share information including the address that CP-E2 is going to connect to. In the TTCN-3 tests, the address is dynamically based on multiple factors at runtime. For Endeavour, it was instead decided to use a single predetermined address for the server socket. Converted tests would still contain the address determined during the TTCN-3 tests so it needed to be replaced. The test conversion process was changed so that at the end of it the XML file is parsed for the field where the address is sent over. If such a field exists in the file, then the address in it is replaced with the hard-coded one, which allows converted tests to connect to the correct address during execution. With a known address bound to the server socket, it is possible to start listening on the socket with <sys/socket.h>'s listen. All of the steps done to set up the server socket so far are done in the SCTP interface's initializer. The code written for setting up the server socket can be seen in figure 12.

Figure 12: The code for setting up the server socket.

```
void Sctp::setupListenerSocket(const char* ip, int port) {
    listenerSocketFd = socket(PF_INET, SOCK_STREAM, IPPROTO_SCTP);
    if (listenerSocketFd == -1) throw std::runtime_error{ "SCTP socket creation failed" };
    sockaddr bindAddr{ addrFromIp(ip, port) };
    if (sctp_bindx_orig(listenerSocketFd, &bindAddr, 1, SCTP_BINDX_ADD_ADDR) == -1) throw std::runtime_error{ "SCTP socket binding failed" };
    if (listen(listenerSocketFd, 1) == -1) throw std::runtime_error{ "SCTP socket failed to start listening" };
}
```

Accepting the client's connection is a bit more involved process as it is necessary to wait for the client to connect before anything else can be done. Having the entire interface wait for the connection to come through would be problematic as the interface would not be able to process other function calls. Having a separate thread for accepting the client prevents this from becoming a problem. The new thread is made at the end of the initializer with std::thread after the server socket is set to listen. In the new thread, the server socket is polled in a loop with <poll.h>'s poll. When the client tries to connect to the server socket the connection is accepted with <sys/socket.h>'s accept and the resulting file descriptor is stored. This file descriptor is a new socket that is connected to the client. When the socket is accepted the polling loop can be exited and the connected socket can, in turn, be polled in another loop for new messages. When there is a new message, it can be received with

<unistd.h>’s read. The messages that are received on the socket are discarded as they are already going to be sent over to Endeavour when CP-E2 sends them with the original sctp_sendmsg. When there are new events on the connected socket and read returns zero bytes, the client has closed its end of the connection. When this happens the connected socket is closed with <unistd.h>’s close. As CP-E2 SCTs sometimes require additional connections the process of accepting new clients and listening to them until they close is looped until test execution ends. The code written for this loop is shown in figure 13.

Figure 13: The code for accepting a new SCTP one-to-one connection and listening to it until it closes.

```
void Sctp::pollSockets() {
    while (not endeavour::isShuttingDown()) {
        pollListenerSocket();
        pollConnectedSocket();
    }
}

void Sctp::pollListenerSocket() {
    std::array pollFds{ pollfd{ listenerSocketFd, POLLIN, 0 } };

    while (connectedSocketFd == -1 && not endeavour::isShuttingDown()) {
        if (poll(pollFds.data(), pollFds.size(), 20) && (pollFds[0].revents & POLLIN)) {
            connectedSocketFd = accept(listenerSocketFd, nullptr, NULL);
        }
    }
}

void Sctp::pollConnectedSocket() {
    std::array pollFds{ pollfd{ connectedSocketFd, POLLIN, 0 } };

    while (connectedSocketFd != -1 && not endeavour::isShuttingDown()) {
        if (poll(pollFds.data(), pollFds.size(), 20) && (pollFds[0].revents & POLLIN) ) {
            receiveAtSocket();
        }
    }
}

void Sctp::receiveAtSocket() {
    // Nothing is done to the received messages here as they are already handled in sctp_sendmsg when they are send by the client.
    std::array<char, 128 * 1024> buffer;
    const auto bytes = read(connectedSocketFd, buffer.data(), buffer.size());

    // Read returns 0 if the other side closes the connection in an 1-to-1 SCPT connection.
    if (bytes == 0) {
        close(connectedSocketFd);
        connectedSocketFd = -1;
    }
}
```

## 4.7 Handling non-notification messages send over SCTP

Running the test on the Endeavour server with the new changes in place resulted in a failure due to timeout and the server output revealed that the socket was receiving messages in a loop and thus needed to be forcefully shut down. The logs showed that everything was going as expected until CP-E2 send a message, E2AP request, over the established SCTP one-to-

one. After that CP-E2 was waiting for a response and as no response was received it continued sending the request repeatedly to the tester. E2AP request and E2AP response are both sent as ASN.1 PER encoded messages. This posed some significant challenges as converting a PER-encoded message to JSON would be a lengthy process. One reason for this is that PER decoding is quite complex, and the scheme used to encode the response would need to be obtained somehow and then included in Endeavour. This would potentially have to be done for a countless number of other E2AP messages and then the schemas would have to be maintained in the future as they might become outdated if the messages they represent change. Because of this, creating a function to manually serialize the message to JSON, like with the SCTP notifications, was not a preferable approach. Thus potential alternatives were explored to see if it could be avoided. There were two internal tools that seemed to offer the desired functionality and some external ones. After some consideration, an internal tool called CASN appeared to be the most suitable one out of the available options so its viability was explored first.

Access had to be requested to use CASN, which took a while to be resolved. While waiting a test was done to see if implementing the E2AP response was the last thing needed for the test at hand to be fully functional in Endeavour. Messages in Endeavour are only serialized for readability and ease of modification and then deserialized back to binary. The only other requirement is for the message to be in JSON for the comparison between expected and real messages to work. Therefore the E2AP response can be saved to a JSON-style string during test conversion in any sort of format, as long as it can be converted back to binary during test execution. In the SCTP serializer, a separate serializer for E2AP protocol data units (PDUs) was created, which is then called in the SCTP interface's sctp_sendmsg. The response is sent as a void pointer from which it can be accessed in its raw binary form. While binary can not be converted directly to string each byte can be first converted to decimal and then added to a JSON array. The number of bytes is known as it is passed to sctp_sendmsg as well. Then the JSON array can be added to a converted test as a string.

During Endeavour test execution the PDU has to be deserialized, so a deserializer for E2AP PDUs was created in the SCTP serializer. To deserialize the string it just had to be parsed and each byte had to be converted back to binary. As the initial void pointer points to multiple

bytes, each byte had to be added to an array in the same order as they were originally in. Std::vector was used instead of a C-style array to comply with modern C++ recommendations (Ottaviano, 2022). Each decimal-represented byte in the string was converted back to binary and then pushed to the vector. The vector's data function returns a pointer to the underlying array which gets implicitly converted to a void pointer when passed to the original sctp_sendmsg. Sctp_sendmsg also needs the message's size which can be obtained from the vector's size as it is equal to the number of bytes in the original PDU.

When a message is sent by the tester during Endeavour test execution, the message's header determines how it is handled. One component of Endeavour allows handling based on the type in the header to be registered. The handling itself is also simultaneously passed to this component in the form of a lambda expression. During Endeavour tests execution when the tester sends a message of the registered type the handling is called and the message's body is passed to the lambda expression. This body is the serialized version of the initial message that can be seen in Endeavour tests and in Endeavour's interface. After the body is deserialized it can be sent to the SUT. To get this process working with E2AP response a field in the header is added indicating that it is of type E2AP PDU when it is added to the converted test. Then in the SCTP interface's initializer handling for messages of type E2AP PDU is registered. In the lambda expression, the body of the E2AP response is deserialized and then send to the SUT through the connected SCTP socket using the original sctp_sendmsg. Std::condition_variable is used to make sure that an SCTP one-to-one connection exists before the message is sent. The thread where the message is actually handled gets blocked until the connection is set up if it is not yet established. The registration of E2AP-PDUs can be seen in figure 14.

Figure 14: The code for registering E2AP-PDU type.

```
endeavourServerConnection::registerJsonMsg("E2AP-PDU", [this](std::string_view header, std::string_view body) {
    const auto buffer = sctpSerializer->deserializeAsn(header, body);

    if (std::unique_lock lock{ unassignedSocketMutex }; connectedSocketFd == -1) {
        unassignedSocketCv.wait(lock, [&]{ return connectedSocketFd != -1; });
    }
    sctp_sendmsg_orig(connectedSocketFd, buffer.data(), buffer.size(), nullptr, 0, 0, 0, 0, 0, 0);
});
```

The serializer created for E2AP responses could also be used to show E2AP requests in the converted tests. However, as the request is not sent by the tester, the process has to be done differently. The request has to be added to the test during test conversion, but there is no need to send it during test execution as the SUT is also going to send it during Endeavour testing. So when a message that is not an SCTP notification is received on the hooked sctp_recvmsg it can be serialized with the same serializer and sent to the XML file. As a received PDU is not sent by the tester there is no need to deserialize at any point. It is added to the Endeavour test just so that Endeavour can compare it against the actual message received during testing.

With the new changes, converting the test with Endeavour added both E2AP PDUs to the test and running the test in the Endeavour server resulted in a pass. Checking the logs confirmed that the test executed completely. While the PDUs could be seen in the converted test, editing them in the interface would have been difficult. A modified PDU would have had to been written manually and then converted to binary before it could have been used in a test. However, the temporary serialization solution accomplished what it was supposed to by showing that with the test at hand there would be no need to account for additional messages after the ASN.1 handling is implemented properly.

## 4.8 ASN.1 encoding and decoding

After access to CASN was granted, its viability could be further examined. The first test was using a GUI-based version of CASN to try to decode the E2AP request and response that appeared in the Endeavour test. The GUI takes PDUs as hexadecimal so the serialized messages from the temporary serialization solution could just be converted to hexadecimal

and used as such. Decoding with CASN worked flawlessly. Next, a meeting with some developers working on CASN was arranged to discuss how CASN could be used in Endeavour. They advised that there was a CASN library that could be used to decode E2AP PDUs directly to JSON-style strings, but unfortunately, it only offered support for decoding. A request would have to be made for the team working on CASN to have this feature added. Before requesting this feature, a small demo was made to test the library's ability to decode E2AP PDUs. And as it also worked well, the library was seen as the best solution for ASN.1 handling. Therefore it was decided to proceed with asking the CASN team to implement the encoding support.

After the CASN team implemented JSON encoding early access to the updated library was received for testing purposes. The JSON encoding worked as expected apart from one specific PDU not encoding successfully. The problem turned out to be caused by a bug and was fixed in a few hours once it was reported to the CASN team. Plugging the CASN library into the previously implemented serializer was easy as the framework for it was already built. The temporary serializer and deserializer functions just had to be overhauled to use the new library through its application programming interface. No changes were necessary to the SCTP interface.

## 4.9 Unit tests

As the bare minimum required functionality for converting and running a single CP-E2 SCT in Endeavour was implemented, it was a fitting time to look back at the code written and refactor it as necessary. However, writing UTs for the SCTP interface and serializer first makes the refactoring process easier as they can be used to reduce the chance of unintended changes being made. So some UTs for both the SCTP interface and serializer were made.

Google's GoogleTest was used for previous UTs in Endeavour and as there was no reason to switch to a different testing framework it was also used for the serializer's and the interface's UTs. Of the two components written the easier one to test is the serializer. The serializer works as a stand-alone piece of code that does not rely on other parts of

Endeavour, unlike the interface which functions in between separate parts of Endeavour. Due to this, the UTs can be straightforward as they do not require any test doubles. Moreover, each different serializer and deserializer function has an obvious path of logic.

To test a successful notification serialization a sctp_assoc_change is created and then serialized. Each value of the serialized association change is compared against the original one to make sure that the notification was correctly serialized. Each serializer also included multiple parts where they could throw an exception if the message being serialized had invalid values or was impossible in some other way. For example, every serializer would throw if the received message was larger or smaller than its reported size. Every possible exception was tested by creating messages that were incorrect in different ways. For the deserializer the tests were similar, but instead of making a real message, a serialized message was created manually. The message is then deserialized and the values of the original and the deserialized message are compared. In the end, UTs were made for each different path of logic in each different serializer and deserializer in the class. Figure 15 shows part of Google Test's output when running the tests for the serializer.

Figure 15: Some of the UTs for the SCTP serializer as seen in Google Test's output.

```
[----------] 3 tests from SctpSacSerializationTests
[ RUN      ] SctpSacSerializationTests.serializeSac
[       OK ] SctpSacSerializationTests.serializeSac (0 ms)
[ RUN      ] SctpSacSerializationTests.sacIsSmallerThanItsType
[       OK ] SctpSacSerializationTests.sacIsSmallerThanItsType (0 ms)
[ RUN      ] SctpSacSerializationTests.sacIsLargerThanItsType
[       OK ] SctpSacSerializationTests.sacIsLargerThanItsType (0 ms)
[----------] 3 tests from SctpSacSerializationTests (0 ms total)

[----------] 3 tests from SctpAsnSerializationTests
[ RUN      ] SctpAsnSerializationTests.serializeAsnErrorIndication
[       OK ] SctpAsnSerializationTests.serializeAsnErrorIndication (0 ms)
[ RUN      ] SctpAsnSerializationTests.serializeAsnEmptyMsg
[       OK ] SctpAsnSerializationTests.serializeAsnEmptyMsg (0 ms)
[ RUN      ] SctpAsnSerializationTests.serializeAsnInvalidMsg
[       OK ] SctpAsnSerializationTests.serializeAsnInvalidMsg (0 ms)
[----------] 3 tests from SctpAsnSerializationTests (0 ms total)

[----------] 8 tests from SctpSerializerTests
[ RUN      ] SctpSerializerTests.notificationIsSmallerThanHeader
[       OK ] SctpSerializerTests.notificationIsSmallerThanHeader (0 ms)
[ RUN      ] SctpSerializerTests.sizeInHeaderDoesNotMatchNotificationSize
[       OK ] SctpSerializerTests.sizeInHeaderDoesNotMatchNotificationSize (0 ms)
[ RUN      ] SctpSerializerTests.serializeWithUnsupportedNotificationType
[       OK ] SctpSerializerTests.serializeWithUnsupportedNotificationType (0 ms)
[ RUN      ] SctpSerializerTests.deserializeAsnErrorIndication
[       OK ] SctpSerializerTests.deserializeAsnErrorIndication (0 ms)
[ RUN      ] SctpSerializerTests.deserializeNotificationWithUnsupportedMsgId
[       OK ] SctpSerializerTests.deserializeNotificationWithUnsupportedMsgId (0 ms)
[ RUN      ] SctpSerializerTests.deserializeNotificationWithUnsupportedMsgName
[       OK ] SctpSerializerTests.deserializeNotificationWithUnsupportedMsgName (0 ms)
[ RUN      ] SctpSerializerTests.deserializeAsnWithUnsupportedMsgName
[       OK ] SctpSerializerTests.deserializeAsnWithUnsupportedMsgName (0 ms)
```

UTs for the interface component are more complicated. The interface needs to interact with other parts of Endeavour, many of which were lacking test doubles. As there was no need to create very in-depth UTs, simple stubs were created as placeholders for proper mocks. Similarly, there was no need to reach 100% test coverage, so it was possible to just create the most important tests, such as checking that all the hooked SCTP functions end up calling their original counterparts at some point. The reason for worse test coverage in the interface compared to the serializer was the time required to implement UTs. Due to the serializer's simplicity reaching full test coverage was relatively effortless, while for the interface just testing the most crucial parts was a difficult and lengthy process.

## 4.10 Final amendments and merging

Trying to convert all of the CP-E2 TTCN-3 tests worked for the vast majority of tests. Looking at the logs of the failing tests revealed two different reasons for failing. Firstly, some tests need to handle an additional SCTP notification: SCTP shutdown event. Secondly, some tests received an SCTP association change with a larger-than-expected size. The fix for the first problem was implementing similar handling for the shutdown event as was done for the association change. The second problem was caused by an unhandled field, sac_info, in SCTP association change serialization. As in many cases the field is unfilled it seemed like there might not be a need for it, but some tests end up filling it. Sac_info is an array with a default size of zero so if it is filled the size of the whole notification increases. As memcpy was used, there were many checks in place to make sure that too many or too few bytes would never be copied. One of these checks caused the tests to fail as the notification could in fact be larger than it is by default and therefore had to be reimplemented in a different way. Although parsing the sac_info required techniques such as bit shifting, implementing its serialization was still mostly about creating JSON key-value pairs based on its contents. With these fixes applied all CP-E2 SCTs could be converted with the Endeavour converter. Additional UTs were created for serializing an SCTP shutdown event and an association change with sac_info.

Trying to run all the converted tests in the Endeavour server resulted in mixed results. Once again the majority of tests worked, but still, six out of the 120 CP-E2 SCTs were failing as they used one additional type of message. These messages are internal TTCN-3 messages that are used to set the state of the tester's SCTP connection. As it was clear that these messages could be implemented in Endeavour, the goals of this work were met. Now there was enough proof to show that all CP-E2 SCTs could be replicated in Endeavour. While making the necessary changes for the last tests to work would have been brief it was not done immediately as another more urgent development task within the Endeavour project took priority. So instead the problem was just reported in Endeavour's development backlog.

As all of the functionality that was asked for was done, the only thing left to do was to merge the feature branch where all the work so far was done to Endeavour's master branch.

Gitlab's merge request feature was used to assist with the merge. First, the feature branch was rebased on top of the master branch. Then other members of the team were asked for a code review. As some of the more complex and error-prone parts were already reviewed and addressed during or after their implementation the problems pointed out in the code review were mostly stylistic. Fixing them was a small effort. Once a commit with the fixes was pushed, the feature branch was merged into the master.

# 5 Conclusion

Overall the aims of the work were met. With the changes made, Endeavour could be used to correctly convert and execute all but six TTCN-3 CP-E2 SCTs. And, as the cause of the last six SCTs' failure was known, more than enough had been done to show how viable Endeavour is for CP-E2 testing. Based on the work done, it can be said that Endeavour can be used in place of TTCN-3 in CP-E2 SCTs, but whether it actually will be is not yet known. The code submitted was also deemed satisfactory and as such would be unlikely to cause any problems, such as technical debt, for the project. While the implementation met all the expectations it could still be developed further. Most obvious of which would be doing the necessary changes for the last six broken tests to work in Endeavour. Secondly, the UT coverage of the SCTP interface could be improved.

The biggest challenge during the development process was understanding the project in the beginning. As the project was in an early state it was rapidly evolving and the documentation was somewhat lacking. Still, the project was not in such an early state that it would have been quick to understand everything going on either.

There were many aspects to implementing the functionality which were not explained for the sake of readability. Many changes had to be done throughout the process as other developers updated how various things were handled. Usually, the changes just meant that some earlier commits had to be re-evaluated and changed. Another constant source of change was refactoring, which was done throughout the process whenever it was deemed beneficial.

## List of references

3GPP. (2015, July 8). *About 3GPP*. Retrieved September 29, 2022, from 3GPP: https://www.3gpp.org/about-3gpp

Ali, I., & Guttman, E. (2018). Path to 5G: A Control Plane Perspective. *Journal of ICT Standardization*, 87-100. Retrieved September 30, 2022, from https://www.riverpublishers.com/journal_read_html_article.php?j=JICTS/6/1/6

Cloudflare. (2020, September 18). *What is the control plane? | Control plane vs. data plane*. Retrieved September 30, 2022, from Cloudflare: https://www.cloudflare.com/learning/network-layer/what-is-the-control-plane/

Fowler, M. (2014, May 5). *UnitTest*. Retrieved September 30, 2022, from martinfowler: https://martinfowler.com/bliki/UnitTest.html

Hamilton, T. (2022, April 30). *What is Component Testing? Techniques, Example Test Cases*. Retrieved June 9, 2022, from Guru99: https://www.guru99.com/component-testing.html

Hamilton, T. (2022, April 30). *What is Software Testing? Definition, Basics & Types in Software Engineering*. Retrieved June 9, 2022, from Guru99: https://www.guru99.com/software-testing-introduction-importance.html

IETF. (2017, October 12). *RFCs.* Retrieved September 29, 2022, from IETF: https://www.ietf.org/standards/rfcs/

Israel, U., & Gardella, M. (2021, Jan 13). *3GPP enabling new monetization capabilities*. Retrieved September 30, 2022, from Nokia: https://www.nokia.com/blog/3gpp-enabling-new-monetization-capabilities/

ITU. (2020). *Introduction to ASN.1*. Retrieved September 29, 2022, from ITU: https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx

McGregor, M. (2020, May 17). *Legacy Code Tips – How to Take Over an Existing Project and its Codebase*. Retrieved June 18, 2022, from freeCodeCamp: https://www.freecodecamp.org/news/taking-over-an-existing-project/

Nokia. (2021, March 18). *Nokia strategy*. Retrieved October 20, 2022, from Nokia: https://www.nokia.com/about-us/company/nokias-strategy-2021/

Ong, J., & Yoakum, J. (2002, May). *An Introduction to the Stream Control Transmission Protocol (SCTP).* doi:10.17487/RFC3286

OSS Nokalva. (2006, March 14). *ASN.1 Made Simple — Encoding Rules*. Retrieved September 29, 2022, from OSS Nokalva: https://www.oss.com/asn1/resources/asn1-made-simple/encoding-rules.html

OSS Nokalva. (2018, September 27). *ASN.1 Made Simple — What is ASN.1?* Retrieved September 29, 2022, from OSS Nokalva: https://www.oss.com/asn1/resources/asn1-made-simple/introduction.html

Ottaviano, L. (2022, February 26). *Three ways to avoid arrays in modern C++*. Retrieved October 5, 2022, from develer: https://www.develer.com/en/three-ways-to-avoid-arrays-in-modern-cpp/

Peterson, L., & Sunay, O. (2020). *5G Mobile Networks: A Systems Approach.* Systems Approach LLC. Retrieved June 18, 2022, from https://5g.systemsapproach.org/index.html

Qualcomm. (2022). *Everything you need to know about 5G.* Retrieved September 29, 2022, from Qualcomm: https://www.qualcomm.com/5g/what-is-5g

Standard C++ Foundation. (2014, February 20). *Big Picture Issues*. Retrieved September 29, 2022, from Standard C++: https://isocpp.org/wiki/faq/big-picture

Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). *The Sockets Networking API: UNIX® Network Programming* (3rd ed., Vol. 1). Addison-Wesley Professional. Retrieved November 1, 2022, from https://learning.oreilly.com/library/view/the-sockets-networking/0131411551/

Stewart, R., Tuexen, M., Poon, K., Lei, P., & Yasevich, V. (2011, December). *Sockets API Extensions for the Stream Control Transmission Protocol (SCTP).* doi:10.17487/RFC6458

TTCN-3.org Editorial Team. (2013, January 23). *About: Introduction*. Retrieved June 9, 2022, from TTCN-3: http://www.ttcn-3.org/index.php/about/introduction

Willcock, C., Deiß, T., Tobies, S., Keil, S., Federico, E., & Schulz, S. (2011). *An Introduction to TTCN-3* (2nd ed.). Wiley. Retrieved November 1, 2022, from https://www.oreilly.com/library/view/an-introduction-to/9780470977897/