Bachelor thesis
Computing Science

Radboud University

Automatic code generation for protocols via Dezyne and Nail

Author:
Pascal Bongartz
s4770986

First supervisor/assessor:
dr.ir. Erik Poll
`erikpoll@cs.ru.nl`

Second assessor:
Prof.dr. Frits W. Vaandrager
`F.Vaandrager@cs.ru.nl`

January 18, 2020

Abstract

Communication protocols are vital for our daily life. These protocols are in every technology we use nowadays. We find them in our messengers like WhatsApp or Signal. They can also be found in the bank cards which use every day. Most of the time, the protocols are implemented hand-written and can thereby be faulty. Most faults in the implementation of a security protocol happen in the parsing of packets. Other mistakes can happen in the execution of the protocol as programmers do not follow the formal specification of a protocol.

In this thesis, we explore the idea of automatically generate the code for protocols, whereby we especially focus on the protocol Transmission Control Protocol(TCP). Therefore, we conduct two separate case studies. The first case study looks at a lightweight version of TCP. After this, we implement the actual protocol. For the code generation, we use the tools Dezyne and Nail. In Dezyne, we implemented the state machine of TCP. This gave us a basic code framework, which then helped us to implement the actual protocol. After we had defined the structure of a packet of TCP in Nail, the tool gave us the code of a parser and a generator for TCP packets.

Based on performing these two case studies, we conclude that it is possible to model protocols like TCP in Dezyne and Nail. Nevertheless, we had problems with the code produced by Nail and it is not always clear how to use the Nail code.

The usage of the tools, Dezyne and Nail, makes the development of protocols like TCP easier. We can design a stateful protocol in Dezyne in many ways which makes it not easy to verify whether the designed model is the right one. Still, Dezyne has many features that help to model a protocol. The modeling process in the Nail format, however, is more straightforward. It is almost a one-to-one translation from the specification to the Nail format.

Another advantage of the tools is the maintainability of a project because we can change for example the structure of a TCP packet at a higher level. Therefore, we do not need to Nail code by hand. Further, Dezyne lowers the likeliness of a bug in the flow of protocol because of the verification process in Dezyne. Nail lowers the likeliness of a bug in the parsing and generates process. Nevertheless, we detect g++ warnings while compiling the case study.

Contents

Chapter 1

Introduction

Many engineering industries such as the automotive industry use models to design complex systems. No one would start to build an automobile before they first divide such a complex construct into different smaller more specialized system models. Through the abstraction of the models, we can easier identify the complex problems of these constructs. Hence, we can use this paradigm during the process of software development. It will help to improve the productivity and reliability of software systems [Sel03]. Possibly, this paradigm may be used to improve the security of software system because much software frequently used today has a security flaw which can be exploited by an attacker.

This holds explicitly for all stateful protocols, for example TCP. The implementation for such protocols is divided in two different layers:

1. These protocols have certain states where specific actions are allowed. Hence, the implementation of such a protocol will benefit from modeling of the protocol. Therefore, an engineer first needs to design the model before he can start to implement the protocol. Most of the time, this approach is combined with automated code generation. This fact also leads to a more secure software system because a human can always make mistakes during the implementation of such protocols.

2. Most of the security flaws do not occur in the way these protocols were designed. The flaws often happen in the handling of the input of the protocol like the packet of a specific message. The parsers which are parsing these packets are error-prone because an attacker can craft the packets himself and can trigger some edge-case of the parsing[BZ14a].

Hence, the research tries to solve these problems by answering the following question:

> Is it possible to automatically generate code for a protocol like TCP with help of Dezyne and Nail?

Therefore, the research focuses on the creation of a prototype that combines the tools Dezyne [dez] and Nail [BZ14a]. Dezyne and Nail deal with different layers of the protocol. These layers corresponds to the listing on page 3.

We use the tool Dezyne to apply the idea of the programming paradigm of model-driven development[1]. Model-driven development defines the technique, which creates running software from the formal specification. Hence in this research, we use Dezyne to model the protocol TCP and generate the code of the protocol.

The second tool, Nail, is an approach for parser generation [BZ14a]. This means if a developer uses such a parser generator, he does not need to write an error-prone code that processes input. If the developer wants to generate these parsers, he has to define the structure of the input.

After design the protocol in Dezyne and specifying the packet structure in Nail, we have to combine both the generated of the tools. This process results into a prototype which we can use to test the quality of both the tools.

This research consists of two case studies. In the first case study, we try to implement the basicTCP(bTCP) protocol. This protocol is a lightweight version of the real-world protocol TCP. This case study is the first try in working with both of the tools. The differences between bTCP and TCP are the header of both protocols and the fixed size of the packages. Additionally, the state machine of bTCP is simpler. Besides the implementation of bTCP, we compare the bTCP implementation with Dezyne and Nail in C++ with a bTCP implementation in Python.

In the second case study, we implement the real protocol TCP. Thereby, we take a look at how we improved the usage of Dezyne and Nail.

We structured both of these case study after the three steps: designing in Dezyne, translate the packet structure in the Nail grammar format and finally combine the generated code to a prototype.

Chapter 2 gives information on the background of the thesis, like a description of the tools. Chapter 3 describes the first case study. Thus the implementation of bTCP and the first reflection on Dezyne and Nail. Chapter 4 illustrates the second case study of the research. Chapter 5 compares this research with related work. Chapter 6 gives an overview on future work to be done on this topic. Finally, chapter 7 concludes the research and discusses the main results of this research.

[1]Model-driven development

Chapter 2

Preliminaries

This chapter describes all relevant background information of this research. Section 2.1 gives an overview of the model-driven development paradigm as well as an overview of automatic code generation. Section 2.2 describes the tools used of this research. In this section, we also discuss some initial experiences of the tools (see sections 3.5 and 4.5 for more discussion of experience in using the tools).

2.1 Automatic code generation and Model-driven development

Model-driven software development aims to generate the source code for a software system completely or partially from a model. This approach aims at developing models of much simpler complexity than the source code. In this case, the programming paradigm DRY(Don't Repeat Yourself) [Wik19] is used. Not only the source code can be generated automatically but also not executables files like tests and the documentation of a software system. Further, it is not always possible to create a suitable abstraction level to describe a specific domain with the means of the respective programming language. Hence language-independent abstractions are created in the form of modeling languages.

These modeling languages can be specially tailored for individual domains. Such languages are referred to as domain-specific language (DSL). Also, the more general modeling language Unified Modeling Language (UML) is used in model-driven development.

There are several advantages to the usage of model-driven development. As the first advantage, problem descriptions are much clearer, simpler and less redundant due to the increased level of abstraction of DSLs. The usage of a DSL not only increases the development speed but also ensures understandable domain concepts within the project [KLM15]. As a second advantage, the separation of the technical level and the domain-oriented model results in a more accessible development of a software system [Wik19].

Nevertheless, there are also disadvantages in the use of model-driven development. The effort of creating a DSL or tailor the UML for a specific target language can be significant, especially for non-trivial project domains. Further, the code generation of these models only results in a framework, which still has to be supplemented by hand with the actual function. Hence sometimes, this makes it hard to get an overview of the needed time of a project [Sel03].

In most cases, model-driven development results in automatic code generation because of the possibility to code on a higher level. Therefore, engineers can focus on designing their projects and do not need to focus on coding. However, a code generator is required. A code generator can be a stand-alone program, but it is also used in a compiler where the generator generates the machine code the compiled source code.

A code generator translates models that are written in DSL or other abstract forms in the chosen target platform.

2.2 Tools

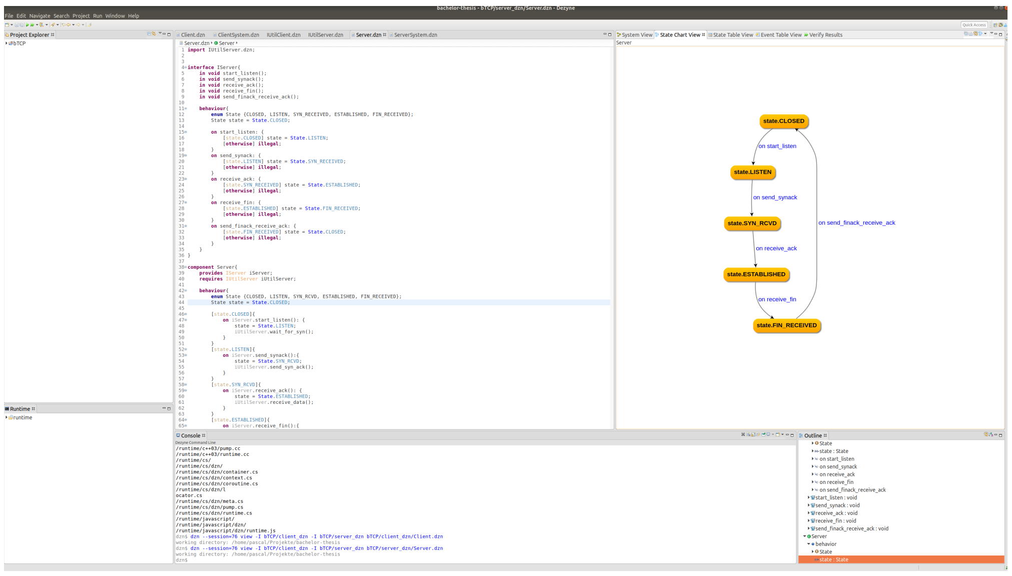
This section describes the tools used within our case studies. Dezyne is a software design tool for modeling-driving development as discussed in section 2.1 and for protocol modeling. The second tool, Nail is a software that generates a parser and a generator for a specified data format. Hence, we use the Nail grammar for modeling the format of the packets of our used protocol and as a result receive code for parsing and generating packets.

2.2.1 Dezyne

Dezyne[dez] is a toolset that enables the usage of the model-driven development paradigm. Dezyne is based on Eclipse IDE which makes the tools more applicable for an engineer. The tool has the same structure as other IDEs(see figure 2.1). Therefore, a programmer can integrate the verification, validation and code generation in the process of designing the model.

Programmers can initially design the system that they require to implement. Therefore, Dezyne uses a newly created domain-specific language(DSL) that is called Dezyne Modelling Language (DML). Sections 3.2 and 4.2 indicate that DML looks similar to other programming languages. However, DML has some differences compared to other modeling languages because of the semantics of DML. The DML has some drawbacks in comparison to other modeling languages like DSL because Dezyne also supports the possibility of checking the designed model[KSHS17]. For example, the DML does not support the construct of time limitations in a model.

As mentioned above, Dezyne provides engineers with the opportunity to formal verify their design model with regards to completeness and correct-



```
import IUtilServer.dzn;


interface IServer{
    in void start_listen();
    in void send_synack();
    in void receive_ack();
    in void receive_fin();
    in void send_finack_receive_ack();

    behaviour{
        enum State {CLOSED, LISTEN, SYN_RECEIVED, ESTABLISHED, FIN_RECEIVED};
        State state = State.CLOSED;

        on start_listen: {
            [state.CLOSED] state = State.LISTEN;
            [otherwise] illegal;
        }
        on send_synack: {
            [state.LISTEN] state = State.SYN_RECEIVED;
            [otherwise] illegal;
        }
        on receive_ack: {
            [state.SYN_RECEIVED] state = State.ESTABLISHED;
            [otherwise] illegal;
        }
        on receive_fin: {
            [state.ESTABLISHED] state = State.FIN_RECEIVED;
            [otherwise] illegal;
        }
        on send_finack_receive_ack: {
            [state.FIN_RECEIVED] state = State.CLOSED;
            [otherwise] illegal;
        }
    }
}

component Server{
    provides IServer iServer;
    requires IUtilServer iUtilServer;

    behaviour{
        enum State {CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, FIN_RECEIVED};
        State state = State.CLOSED;

        [state.CLOSED]{
            on iServer.start_listen(): {
                state = State.LISTEN;
                iUtilServer.wait_for_syn();
            }
        }
        [state.LISTEN]{
            on iServer.send_synack():{
                state = State.SYN_RCVD;
                iUtilServer.send_syn_ack();
            }
        }
        [state.SYN_RCVD]{
            on iServer.receive_ack(): {
                state = State.ESTABLISHED;
                iUtilServer.receive_data();
            }
        }
        [state.ESTABLISHED]{
            on iServer.receive_fin():{
```

```
/runtime/c++03/pump.cc
/runtime/c++03/runtime.cc
/runtime/cs/
/runtime/cs/dzn/
/runtime/cs/dzn/container.cs
/runtime/cs/dzn/context.cs
/runtime/cs/dzn/coroutine.cs
/runtime/cs/dzn/l
ocator.cs
/runtime/cs/dzn/meta.cs
/runtime/cs/dzn/pump.cs
/runtime/cs/dzn/runtime.cs
/runtime/javascript/
/runtime/javascript/dzn/
/runtime/javascript/dzn/runtime.js
dzn$ dzn --session=76 view -I bTCP/client_dzn -I bTCP/server_dzn bTCP/client_dzn/Client.dzn
working directory: /home/pascal/Projekte/bachelor-thesis
dzn$ dzn --session=76 view -I bTCP/client_dzn -I bTCP/server_dzn bTCP/server_dzn/Server.dzn
working directory: /home/pascal/Projekte/bachelor-thesis
dzn$
```

Figure 2.1: Structure of Dezyne

ness prior to implementation. Besides formal verification, Dezyne additionally discovers the errors of the model and displays them to the programmer.

There are several things that a user of Dezyne can verify and validate with the help of Dezyne. What the user can verify and validate depends on the different components of the model(see sections 3.2 and 4.2). Dezyne allows user to verify and validate the following five aspects:

1. Check for deadlock
2. Check for livelock[1].
3. If the model is deterministic.
4. Check for illegal behavior in a component. This illegal behavior is defined in an interface(see sections 3.2 and 4.2).
5. Check for right behavior of component based on the required interface.

With these five checks, a user has a good overview of whether his model is properly designed or not.

Furthermore, Dezyne is able to simulate and validate a specified model. The engineers can simulate the behavior of their design and can immediately validate whether the design functions in the correct way. Thereby the programmers can inspect the trace of the simulation and identify the errors inside the DML.

[1]https://en.wikipedia.org/wiki/Deadlock

Ultimately, Dezyne's primary feature for this paper is code generation. After verifying and validating his designs, an engineer can easily generate the source code for this model. This code can be integrated into already existing code or it can be used as a base of a new project. Nevertheless, extern functions like socket setup need to be programmed by an engineer.

After the initial setup of Dezyne on our computers, we completed the tutorial and discovered the tool, which allowed us to start to design our first case study 3.

2.2.2 Nail

Nail is a so-called parser generator [comb]. Hence, Nail generates parser based on a formal specification of the code of a parser. These parser generators are often used to handle syntactic analysis, which means the generators are a subprogram of a new compiler. Besides their usage in compilers, we also use parsers in all sorts of applications. For example, a generator produces the source code for a parser based on an input grammar, which defines the syntax of a new programming language. This parser can then parse this new programming language, which is essential for the compiling process of the language.

Using a parser generator in the implementation of a protocol eliminates one of the biggest security flaws in protocols: a hand-written parser. A hand-written parser for protocols is always vulnerable for buffer overflows and improper input validation [BHH+17]. We validate input because of handcrafted malicious input. This input can lead to misbehavior in executing code and leak critical data for an attacker.

Most of the parser generators only focus on the part of parsing and did not consider the contrary direction. Nevertheless, Nail produces code in both directions. Hence, the generated code of Nail can be implemented to parse certain syntax, but it also can be used to generate an output of this certain syntax. This property is essential if we want to integrate a parser generator in the implementation process of protocols, especially in the process of security protocols.

Parsing and generating are essential for the implementation of protocols because a server and a client often need to exchange some packets before they establish a connection between them. Also when transferring data via a protocol, the packets have a given grammar/structure. Hence, it is valuable if the used parser generator can produce code, which can parse and generate packets of a certain grammar/structure.

Another feature of Nail is the possibility of handling offset fields and checksum elegantly. Thereby, Nail introduces two abstractions: The dependent field helps to represent fields in protocols that are dependent on the values of the rest of the packet; transformations allows programmers to work on raw data of an important packet for computing checksum.

However, at the beginning using Nail is not that straightforward. About the tool, we discovered several papers([BZ14a], [BZ14b],[BZ15a]) and not all of them contain an explanation of the grammar format that Nail uses. But in the paper [BZ14a], we found a table of the grammar format of the tools, which eases to understand the tool. Further, we found some examples of Nail grammar formats of well-known protocols, which provide the first impression of the this grammar format. Sections 3.3 and 4.3 feature an example of the Nail grammar format.

Furthermore, it is beneficial to take a look at the examples of the GitHub page [BZ15b]. These examples give several usages of the Nail grammar. On top, the examples provide on an overview of how to implement the automatically generated code of Nail in hand-written code.

After overcoming the initial problems, the generated Nail code is clear to use in the case study. For more reflection see sections 3.3 and 4.3.

Another drawback of Nails the lacking up-to-dateness of the tool, with the last commit on GitHub dating back to 2015. Furthermore, as we took a look into automatically generated code, we found some comments with Todos which suggest some updates in the structure of the Nail code and in the some of the functionality of Nail .

Chapter 3

Case study: bTCP

This chapter describes the first case study of this paper. This case study looks at the implementation of bTCP(see section 3.1) with the help of Dezyne and Nail. In sections 3.2 and 3.3, we describe the usage of both tools and the way how we implemented bTCP using these tools. Section 3.4 gives an overview of how to put together the generated code of Dezyne and Nail. Finally, section 3.5 compares the implementation of bTCP with the help of the tools and the implementation of bTCP in Python, and also describes the experience of working with the mentioned tools in practice.

3.1 bTCP

As the first case study, we choose a simple protocol to implement in order to get to know both the tools Dezyne and Nail. We decided to use the bTCP protocol. We already implemented a version of this protocol the bachelor course Networks and Disturbed Systems[1] where we realized the protocol in Python. During the course, we needed to implement, how to parse and generate the packets of the protocol ourselves. Further, we needed to construct the state-machine of bTCP.

bTCP stands for basicTCP because this version of the protocol consists of a more simplistic header and does not use the idea of congestion control. Another simplification compared to the real TCP is the specified length of the packet that will be sent. A packet of bTCP is always 1016 bytes long. The first 16 bytes represent the header of the packet. The remaining 1000 bytes are reserved for the sent data.

[1]https://www.ru.nl/courseguides/science/vm/osirislinks/ibc/nwi-ibc021/

The header of bTCP looks as follows:

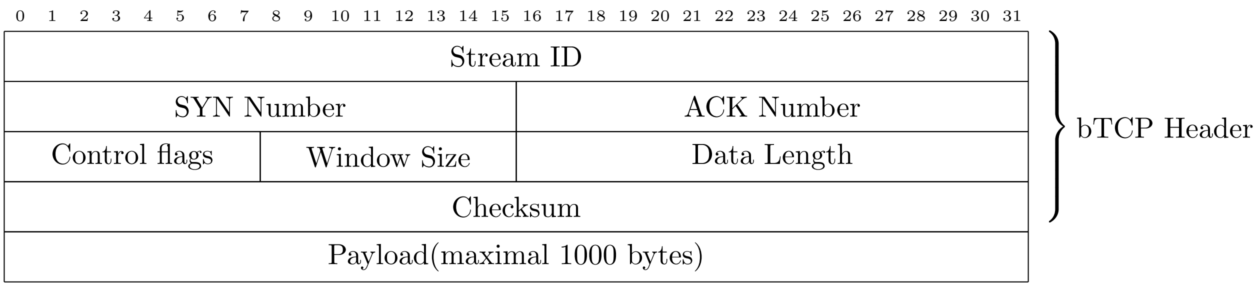


Figure 3.1: The bTCP Header

The fields are defined the following way:

1. StreamID (32-bit): A unique identifier given to each bTCP stream. Used to differentiate packet of different origin and destination.
2. SYN Number (16-bit): Used to order packets in a given bTCP stream.
3. ACK Number (16-bit): Used to acknowledge received packets.
4. Control Flags(8-bit): Contains the flag state of a given packet.
5. Window Size (8-bit): Defines the number of packets allowed in transit with a maximum of 255.
6. Data Length (16-bit): Defines how much of 1000 bytes are data.
7. Checksum (32-bit): A checksum computed over the header and data.
8. Payload (max. 1000 bytes): The data of the packet.

Additionally, the checksum is a 32-bit long due to the usage of the CRC32(cyclic redundancy check) algorithm.

The goal of the original implementation was to make an implementation that is reliable. Hence, the client needs to maintain track of the packets that were already acknowledged by the server. The client should retransmit the packet after some timeout, that are unacknowledged by the server. Therefore, the client needs to keep track of the ACK numbers. Moreover, the server should be capable to reassemble out-of-order packets.

Another part of the bTCP protocol is the usage of flow control. The idea of flow control is that the client and the server agree on a specific window size. The window size specifies the maximum number of packets that the client can send before the server acknowledges one of these packets.

Furthermore, the flags are simple to use because they are just an 8-bit integer. Hence, we can define one 8-bit integer for one flag. For example, the number one equals the flag FIN.

As above-mentioned, the initial task was to define a state-machine of this protocol. The state machine of the server and client are presented in figure 3.2 and figure 3.3.

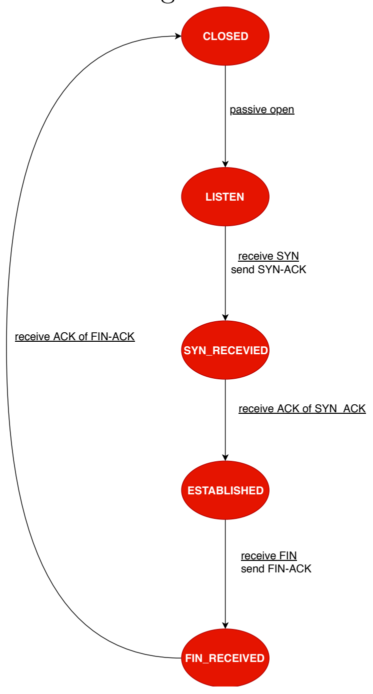


Figure 3.2: State machine of the bTCP server

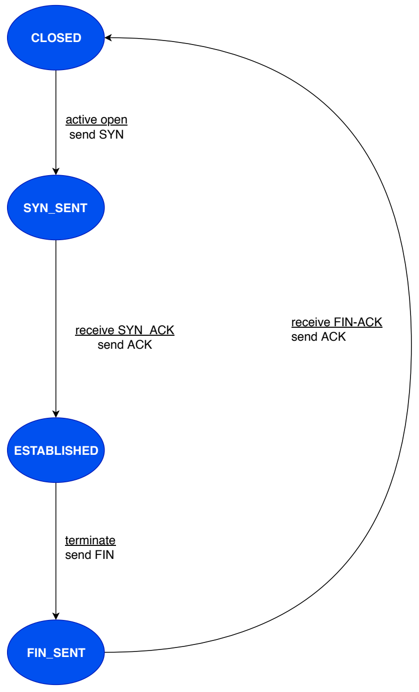


Figure 3.3: State machine of the bTCP client

The initial state of the server is the `CLOSED`. Therefore, the server cannot establish a connection with a client. In the state `LISTEN`, the server sets up its structure and can receive packets from clients. The server waits for a packet with SYN to continue. After receiving the SYN packet, the server sends an SYN-ACK packet and transits to `ESTABLISHED` state. In this state, it waits for the acknowledgment of the SYN-ACK packet. This step concludes the opening handshake, and the server can now process incoming packets with data. The data transmission continues until the server receives a FIN packet. The server advances into the `FIN_RECEIVED` state and sends a FIN-ACK packet. At that point, it waits for an acknowledgment of this packet. After receiving this packet, the connection between server and client is terminated and the server is back in its initial state.

The client shares the initial state with the server. In the state `CLOSED` of the client, the client cannot send any packets. To send packets, the client needs to open the connection wit the server actively . Hence, the client sets up its structure up and transmits the first packet. This packet is an SYN packet that starts the connection between client and server and gets the client to `SYN_SENT` state. In this state, the client waits for an SYN-ACK packet that acknowledges its SYN packet. This packet is again acknowledged by the client, which concludes the opening handshake and the client transits to the `ESTABLISHED` state. At this point, the client starts sending packages with data to the server. This procedure happens until the client delivered all the data and sends a FIN packet. At this point, the client is in the `FIN_SENT` state, and wait for FIN-ACK of the server. Finally, the client acknowledges this packet, sends an ACK packet and goes back to its initial state.

3.2 bTCP in Dezyne

The usage of Dezyne in the process of software development is automatically component-based. Therefore, every part of a system represents a component which defines a certain functionality. To define the functionality, a component can use the functionality offered by another component. The offered functionality of a different component is defined in the interface of the component.

Hence, it is important to initially specify an interface of a component first and consecutively use the component in other components. In the case of the server structure, this fact is not that important because the server component is already the highest in the whole system.

Nevertheless, in Dezyne Modelling Language (DML) an interface provides the events, their direction(input/output) and optionally their parameters. Besides the definition of the events of a component, an interface specifies the externally visible behavior of the component that will provide an implementation of the interface. In this context, behavior means which events (or method calls) can be handled at which stages in the execution process.

```
import IUtilServer.dzn;


interface IServer{
    in void start_listen();
    in void send_synack();
    in void receive_ack();
    in void receive_fin();
    in void send_finack_receive_ack();

    behaviour{
        enum State {CLOSED, LISTEN, SYN_RECEIVED, ESTABLISHED, FIN_RECEIVED};
        State state = State.CLOSED;

        on start_listen: {
            [state.CLOSED] state = State.LISTEN;
            [otherwise] illegal;
        }
        on send_synack: {
            [state.LISTEN] state = State.SYN_RECEIVED;
            [otherwise] illegal;
        }
23          on receive_ack: {
24              [state.SYN_RECEIVED] state = State.ESTABLISHED;
25              [otherwise] illegal;
26          }
27          on receive_fin: {
28              [state.ESTABLISHED] state = State.FIN_RECEIVED;
29              [otherwise] illegal;
30          }
31          on send_finack_receive_ack: {
32              [state.FIN_RECEIVED] state = State.CLOSED;
33              [otherwise] illegal;
34          }
35      }
36  }
37
38  component Server{
39      provides IServer iServer;
40      requires IUtilServer iUtilServer;
41
42      behaviour{
43          enum State {CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, FIN_RECEIVED};
44          State state = State.CLOSED;
45
46          [state.CLOSED]{
47              on iServer.start_listen(): {
48                  state = State.LISTEN;
49                  iUtilServer.wait_for_syn();
50              }
51          }
52          [state.LISTEN]{
53              on iServer.send_synack():{
54                  state = State.SYN_RCVD;
55                  iUtilServer.send_syn_ack();
56              }
57          }
58          [state.SYN_RCVD]{
59              on iServer.receive_ack(): {
60                  state = State.ESTABLISHED;
61                  iUtilServer.receive_data();
62              }
63          }
64          [state.ESTABLISHED]{
65              on iServer.receive_fin():{
66                  state = State.FIN_RECEIVED;
67                  iUtilServer.wait_for_fin();
68              }
69          }
70          [state.FIN_RECEIVED]{
71              on iServer.send_finack_receive_ack(): {
72                  state = State.CLOSED;
73                  iUtilServer.receive_final_ack();
74              }
75          }
76      }
77  }
```

Code fragment 3.1: bTCP server in Dezyne

As we started to model the bTCP state-machines(figures 3.2 and 3.3), we first needed to define an interface for the server(see code fragment 3.1, lines 4-36). The definition of an server interface eases implementation of the server structure in Dezyne. The lines 38-77 describes the component with describes the server.

In the following listing, we giving a high level description of the interface and the component of the server which can be seen in the code fragment 3.1:

Interface(lines 4-36): As mentioned above, you see the interface of the server (IServer) in code snippets 3.1 in the lines 4-36. In lines 5-9, we define all the methods, which the providing component needs to specify. The names of the methods following the state machine of the server 3.2.

Following this, we define the `behavior`-block of the interface. The block echoes which of the methods are allowed in a particular state. The states are visible in the line 12. Thereto, we describe the states in an enumeration. The names of the states are equal to names in the state machine of the server 3.2.

Further, line 13 displays the initial state of the model in the Dezyne environment. Next, we start by defining which functions can be executed in which states. For example, the `receive_fin` method is only legal if the state is in the `ESTABLISHED` state. In all other states, it is illegal to call this function. If a part of the system still tries to call the function, the program terminates.

Component(lines 38-77): At line 40, we define that the server component requires another component namely the `UtilServer` component. This component provides the interface `IUtilServer`(see the code fragment A.1). The interface only defines the signatures of the methods because we only can defines the bodies of these function in C++. We only need the `UtilServer` component to use the functionality of the `IUtilServer` interface in the `Server` component.

In line 42, we start defining the `behavior`-block of the component. The beginning of this mostly looks identical to the interface. Nevertheless, the definition of the functions looks modified. We use the states as a switch-block. For the control flow of the system. For example, the server stands in `LISTEN` state, then we merely define the actions of the method `send_synack`. All other methods of the interface are illegal in this state, therefore it is not necessary to define actions for these methods. Further, the body of these functions is defined in the same way. At the start, we change the state to the following state of the state machine. Second, we call a function of `UtilServer` component.

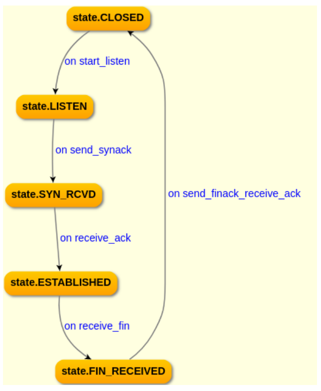


Figure 3.4: State machine of the bTCP server

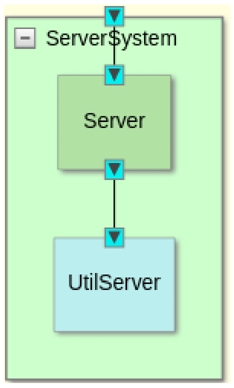


Figure 3.5: System overview of the bTCP server

After modeling the component `Server` of the code fragment 3.1, Dezyne gives us the states machine in figure 3.4. This state machine is equal to the state machine in figure 3.2 on page 12.

In figure 3.5, Dezyne shows us the complete server system. The system is a simpler system because it only contains two component. We connect the system with outer world via the `Server` component. Hence, we can call the function of `Server` directly. The blue color of the component `UtilServer` indicates that this component needs to be written by hand.

Last not but least, we need to connect the `Server` and the `UtilServer` component into a sound system. The code 3.2 defines this connection. Therefore, we create a new component, the `ServerSystem` wh9ich serves the purpose to refer to other components and connects the interface of them. To connect the interfaces, we need to bind, with <=> operator, each `requires` interface instance with the corresponding `provides` interface instance.

```
import IUtilServer.dzn;
import Server.dzn;

component ServerSystem{
    provides IServer iServer;

    system{
        Server server;
        UtilServer utilserver;

        iServer <=> server.iServer;
        server.iUtilServer <=> utilserver.iUtilServer;
    }
}
```

Code fragment 3.2: bTCP server system in Dezyne

After finishing the modeling process, we can verify our model in Dezyne. Besides verification, we also can simulate the behavior of our model and check for modeling mistakes. Verifying our model helped us a lot because it would spot mistakes that we did not see. Furthermore, Dezyne automatically verifies the model if we wanted to generated the code of the model. Simulating the behavior of the model makes it easier to find the mistakes in the model.

The client system is not displayed in this section because the extensive explanation of the Dezyne files and the implementation of the client is the same as the implementation of the server. For convenience, you can find the client Dezyne code in the Gitlab repository[2].

3.3 bTCP in Nail

As we already stated, figure 3.1 on page 11 shows the bTCP header. The bTCP header is not as complex as the real TCP header because most of the fields in the header are simple bytes field. The interesting fields are the control flag, the data length, and the checksum field.

The control flag field is interesting because we can define the value of the flags.

Besides the control flag field, the data length field is interesting because the field reflects the length of the whole packet, which we want to send.

Further, the checksum field is interesting because the checksum is computed over the header bytes and the payload bytes. Therefore, it is more complicated to compute the value of the checksum.

The Nail grammar format has a basic format. You can see the basic format in the code fragment 3.3. To the `btcp` parser, we assignment the rule between the curly brackets. A parser can consist of some basic formats and more sophisticated structures like dependent fields, input streams, and transformations.

One of the basic building blocks of the Nail grammar is signed or unsigned integer with a maximal length of 64-bits. The grammar can also constrain these integers. For example, an integer can only contain a range of integers or a specific number. Additionally, the grammar supports the repetition of other building blocks. Further, it is possible to model structures in Nail grammar. Another building block that is supported by the grammar is constants value. It is further possible to define a structure that consists of a `select`. This `select` behaves like `switch`-statement in a programming

[2]https://gitlab.science.ru.nl/pbongartz/bachelor-thesis/tree/master/bTCP

language. You give the `select` parser a value and then it decides which next parser rule is executed. Moreover, the Nail grammar format supports to choose between two other formats. We also can define optional buildings blocks. To use all the building blocks, we can reference different building blocks. For more explanation, see [BZ14a].

```
1  control_flags = {
2      flag uint8 | 1..6
3  }
4
5  btcp = {
6      streamid    uint32
7      syn_number    uint16
8      ack_number    uint16
9      flags        control_flags
10     window_size    uint8
11     @data_length    uint16
12     @checksum    uint32
13     transform crc32_checksum($current @checksum)
14     payload n_of @data_length uint8
15 }
```

Code fragment 3.3: bTCP packet in Nail grammar

The code 3.3 reflects the bTCP header in figure 3.1. The definition of the bTCP header begins in line 5 with the assignment of the parser to the name bTCP. The first, second and fourth fields are very plain unsigned integers of a specified size. The size of in Nail grammar corresponds with the size in the bTCP header.

The `flags` field references to a second parser that only parses the field of the control flags. In the control flag parser, we only define one format which is a restricted unsigned integer.

The `data_length` and `checksum` fields are also definitions of an unsigned integer, but we use the at-sign as a reference to use both these fields in the following fields.

The last two definitions are the interesting part of the bTCP header in Nail grammar. We define the payload of the packet as a number of unsigned 8-bit integers. The number is specified by the data length field because this field specifies the size of the packet. The last definition uses the format of transformations. Therefore, a programmer himself needs to implement the here defined function. In this case, we have to program `crc32_checksum` in C++. The interface of the method is already given. You can find the implementation this function in the appendix part A.2.

3.4 Putting it together

Figure 3.6 describes the directory structure of our bTCP project. The subdirectories `client_dzn` and `server_dzn` contain the modeled ClientSystem and ServerSystem of section 3.2. The `btcp.nail` file contains the bTCP header in Nail grammar format which we defined in section 3.3.

```
bTCP
├── Makefile
├── server (executables)
├── client (executables)
├── btcp.nail
├── lib(Dezyne library)
├── lib_gen
│   ├── btcp.nail.cc/hh
│   └── checksum.cc
├── nail(Nail library)
├── client_dzn
│   ├── Client.dzn
│   ├── ClientSystem.dzn
│   └── IUtilClient.dzn
├── server_dzn
│   ├── Server.dzn
│   ├── ServerSystem.dzn
│   └── IUtilServer.dzn
├── src_client
│   ├── Client.cc/hh
│   ├── ClientSystem.cc/hh
│   ├── IUtilClient.cc/hh
│   ├── UtilClient.cc/hh
│   └── main_client.cc
└── src_server
    ├── Server.cc/hh
    ├── ServerSystem.cc/hh
    ├── IUtilServer.cc/hh
    ├── UtilServer.cc/hh
    └── main_server.cc
```

Figure 3.6: Directory structure of bTCP

🟧: automatically generated code	🟫: Dezyne models & Nail grammar
🟦: self-written code	⬛: other

The directories `lib` and `nail` contain the libraries of both the tools Dezyne and Nail. We need to copy the Nail library in the project ourselves. We create the Dezyne library with the command `make runtime` in the Makefile in the appendix A.3.

With the command `make generate_all` of the Makefile, all files with the color orange in figure 3.6 are automatically generated. The Dezyne and Nail automatically generate these files. Hence, they are not pretty formatted. Nevertheless, you can still find them in the Gitlab[3] project of the thesis.

The cyan files in figure 3.6 are the program code we have to write ourselves. In the following we list them:

- The files `main_server` and `main_client` starting the server/client. Additionally, they initialize the Dezyne system.
- The files `UtilServer.hh/cc` and `UtilClient.hh/cc` are also handwritten as we mentioned in section 3.2. These files consist of the socket setup and some other things.
- As we already mentioned in section 3.3, we need to program the `crc32_checksum` function. Therefore, we create the file `checksum.cc`. You can see the content of the file in the appendix A.2.

As we already mentioned above, we need to write a main function which executes the Dezyne code. We have to this for the server and the client. In this section, we only explaining the file at example of the server, but you find the client part in the Gitlab repository[3].

```
#include <iostream>
#include <string>

#include <dzn/runtime.hh>
#include <dzn/locator.hh>
#include "ServerSystem.hh"

int main(int argc, char* argv[]){
    dzn::locator loc;
    dzn::runtime rt;

    loc.set(rt);

    ServerSystem serverSystem(loc);

    serverSystem.check_bindings();

    serverSystem.iServer.in.start_listen();
    serverSystem.iServer.in.send_synack();
    serverSystem.iServer.in.receive_ack();
    serverSystem.iServer.in.receive_fin();
    serverSystem.iServer.in.send_finack_receive_ack();
}
```

Code fragment 3.4: main_server.cc file

[3]https://gitlab.science.ru.nl/pbongartz/bachelor-thesis/tree/master/bTCP

The code fragment 3.4 represents the main file where all parts of the server are called. To run a Dezyne system in C++, the system needs the runtime and locator of Dezyne because the generated code of Dezyne uses functions from these header files. After we include the header files into our project, we can make an instance of both the runtime and the locator.

The next step is to create an object of the ServerSystem in line 11. After this step, we use the `check_bindings` function of Dezyne. The usage of the function is recommended because it checks if all ports are bounded properly.

Finally, we can start to use the defined methods of the server. In this version, the server can only handle one user and one session because we only call all the server methods one time(lines 15-19). The `main_client.cc` has a similar structure as the code 3.4. You can find the `main_client.cc` file in the Gitlab repository[4] project of the thesis.

```
1   /*generates a bTCP packet based on the input and directly send the packet*/
2   void genPacket(NailArena *arena, int streamID, int synNr, int ackNR, int flags, int
    ↪  window_size, int data_length,
3           std::vector<unsigned char> payload) {
4       //C++ struct of the bTCP packet
5       btcp packet;
6       NailOutStream out;
7       NailOutStream_init(&out, 2000);
8       memset(&packet, 0, sizeof(packet));
9       packet.streamid = streamID;
10      packet.syn_number = synNr;
11      packet.ack_number = ackNR;
12      packet.flags.flag = flags;
13      packet.window_size = window_size;
14      packet.payload.count = data_length;
15      narray_alloc(packet.payload, arena, packet.payload.count);
16      for (unsigned int i = 0; i < packet.payload.count; i++) {
17          packet.payload.elem[i] = payload[i];
18      }
19      //main generate function of Nail: generate bTCP packet in arena
20      if (gen_btcp(arena, &out, &packet) != 0) {
21          printf("gen packet");
22          exit(0);
23      }
24      //getting bTCP out of the arena, we can send buf
25      const unsigned char *buf = NailOutStream_buffer(&out, &len);
26  ...
```

Code fragment 3.5: Generating a bTCP packet and sending packet

The code 3.5 shows how we can generate a new bTCP packet. To begin with, we create an instance of the bTCP packet inline 3. We also need a `NailOutStream` that we use for receiving the generating packet out of the `arena`. The variable `arena` is one of the unique parts of the Nail tool. The further step is to give the bTCP object the attributes of the packet. We can accomplish this easily by just assigning the new value to the attributes of the bTCP object(lines 7-16). Therefore, the payload field is special because we loop through the payload and assign the single values of the payload.

[4]https://gitlab.science.ru.nl/pbongartz/bachelor-thesis/tree/master/bTCP

Now, we call the `gen_btcp` function. This function creates the packet in the arena. With the function inline 23, we stream the bytes object of the packet out of the arena and write the variable `buf`. Finally, we can forward this to the client which is not shown the code fragment 3.5.

```
uint8_t packet[1016];
ssize_t packet_size;
NailArena arena;
jmp_buf err;
...
   //receive packet from the socket
    packet_size = recvfrom(sock, packet, sizeof(packet), 0, (struct sockaddr *)
    ↪  &rem_addr, &addr_len);

    if (0 != setjmp(err)) {
        printf("OOM\n");
        NailArena_release(&arena);
    }
    //initialize arena(=parsing memory) and stream(=getting packet in bytes into
    ↪  arena)
    NailArena_init(&arena, 2000, &err);
    NailMemStream packet_stream(packet, packet_size);

    //main parser function of Nail, returns a bTCP struct
    message = parse_btcp(&arena, &packet_stream);
    if (!message) {
        printf("Invalid packet; ignoring\n");
        exit(0);
    }
...
```

Code fragment 3.6: Parsing a bTCP packet

The code 3.6 explains the process of parsing a bTCP. Before parsing a packet, we declare some help objects. As in the generating part, we need a `NailArena` and a `NailOutStream`. Additionally, we need `jmp_buf` object. This object helps with memory errors of Nail. Now, we receive the packet from the input socket and stream this packet in the `NaimMemStream` object `packet_stream`. We feed this object into the `parse_btcp` function together with the defined `NailArena`. The function returns a bTCP object and we use the object onwards in our program.

3.5 Reflection

This section gives a comparison between the C++ implementation of bTCP built using Nail and Dezyne and the Python code of the bTCP implementation(in 3.5.1). Also, the section details the experiences of working the first time with Nail and Dezyne(in 3.5.2 and 3.5.3). Section 3.5.1 describes problems which occurred by combing Dezyne and Nail code.

3.5.1 Comparison between Nail + Dezyne and Python

When we want to compare the implementation of bTCP in C++ with the help of Dezyne and Nail and the implementation in Python that I made earlier as part of the course Networks and Distributed Systems, we can take a look at the lines of code both implementation. Nevertheless, we look only at the hand-written parts and not the header files of the C++ implementation(See 3.6).

The server in C++ consists of 415 hand-written lines of code. The `main_sever` file contributes 20 lines of code and the `UtilClient` file produces 395 lines of code. Additionally, we could count the lines of the `checkusm` file but we cannot compare this number with the Python implementation. In the Python version of bTCP, the checksum is computed with a library and only uses one method call. In comparison, the server in Python consists only of 318 lines of code. This is a difference of 100 lines of code. This discrepancy may raise questions. Why is an implementation with the help of two tools longer than an implementation without help? The reason for the discrepancy lies in the properties of the programming languages and how we can use the tools.

C++ has more verbose syntax than Python. In C++, all methods need curly brackets. Hence, a method in C++ always has two more lines than the counterpart in Python. Additionally, we have to declare some variables beforehand in C++, which is in Python not needed. Also, you have to manually handle memory in C++.

Another reason why the C++ implementation has more lines of code is the usage of generated code. As you could see in code snippets 3.5 and 3.6, using the Nail code needs some preparation. You always have to declare the arena and output stream before parse or generate a packet. Further, you have to release these objects because they allocate memory which can lead to a segmentation fault.

Another way to compare both of these implementations is the performance of them. Hence, we would compare the execution time and memory usage of both implementations. This could lead us to a decision which one of the implementations should further be used. Nevertheless, this comparison is valuable, but we already know the outcome. Python is not fast enough for communication protocol like bTCP because Python is a not compiled language.

3.5.2 The Experience using Dezyne

This case study represents the first try for us to program a protocol with the help of Dezyne. As we mentioned in section 2.2.1, at the beginning the usage of Dezyne was not trivial. The tool has its own modeling language (DML) which we first need to understand. Besides the new language, it was the first attempt to model the system before we start directly to program the system. After we studied the tutorials on Dezyne webpage[dez], the tool became more understandable for us.

For this case study, it was convenient that we already created the state-machines for the sever 3.2 and client 3.3. The state-machines assisted us to design because we did not need to think about the flow of the protocol. We could concentrate on the modeling process that helped us a lot.

We only encountered one problem during the design bTCP in Dezyne. As you can see in code 3.1, we could use the keyword `otherwise` in the specification of the interface. But in the beginning, we started to define the action for every state. Afterward, we discovered the keyword `otherwise` that made the definition of the interface more clearer.

After we defined and verified bTCP in Dezyne (see section 3.2), we did not generate the code via the Makefile command because this process was new for us. Nevertheless, we discovered an example of a Makefile for a Dezyne project and modified the file to our purpose(See A.3). With this Makefile, we made every step of generating the code of Dezyne and compiling the code automatically. Hence, in future projects, we only need to specify the model and then create the Makefile that will set-up the rest for us.

Additionally, we were surprised by how many lines of code Dezyne creates based on the model files. For example, the server in figure 3.1 in DML has 77 lines of code. Dezyne creates for this server in C++ 365 lines of codes. We would expect Dezyne would create more lines of C++ code because the size of model.

3.5.3 The Experience using Nail

We also used Nail for the first in this case study. Hence, we need some time in the beginning as we discussed in section 2.2.2.

To begin with, we did not know how to model the bTCP header(3.1) in the Nail grammar format because of the inadequate documentation of Nail. Nevertheless, we solved this problem quickly and model the header in Nail(see figure 3.3). We think it is simple to model a packet because the syntax of Nail is very understandable. Further, the structure of the bTCP header is not very complex, which makes the modeling simpler.

As you can see in the Makefile A.3, we automated the code generation of Nail code. Hence, we only wrote the Nail grammar for a bTCP packet in our favorite text editor, save the file with the nail format and call the right Makefile rule.

During the implementation of self-written function `crc32_checksum`, another problem occurred. We did not how to code the method. After some searching in the examples of Nail, we found an example implementation of such a method.

Furthermore, we needed to accustom to the arenas which are provided by Nail. We had to specify these arenas by every use of parsing or generating a packet. But after some time, the process becomes natural.

Additionally, we were surprised by how many lines Nail produces from the specified input grammar. The grammar 3.3 has only. Nail creates based on this grammar 396 lines of code. To the 396 lines, we need to add 30 lines of code because we needed to program the checksum computation ourselves.

During the compiling of the process of the bTCP project, another problem occurred. g++, the C++ compiler used of our project, produces three different warnings. We cannot neglect one of these warnings because of the possibility of a security flaw. Therefore, we discuss these warning below:

-Wsign-compare: This alert warns us for a comparison between signed and unsigned integer expressions. This warning is very dangerous for the source code of Nail because this comparison can cause several problems. One problem is that C++ provides implicit type conversions between signed and unsigned values. For example, when a signed integer is converted to an unsigned integer, the bit pattern ist left alone and the vale changes correspondingly.

This warning occurs in a for-loop generated by Nail. Therefore, the signed integer is always converted to an unsigned integer. The conversion can cause not desired behavior.

The comparison between signed and an unsigned integer is also a security problem because it might cause a buffer overflow or it might cause bug, but it is not a buffer overflow.

-Wunused-variable and -Wunused-function: This first alert warns us for an unused variable and second alert warns us for an unused function. These warnings are not dangerous because they do not trigger any security flaws. Often such variables and functions are remains of debug output or just old functions that are not used anymore. As we already mentioned, Nail is not the update-to-date and only project of a Master-thesis. Hence, the creator of Nail could forget these unused parts.

It can be important to fix these warnings in future research(see chapter 6) because besides the warnings Nail is a simple tool. The tool has a simple grammar that results in generated code which can be efficiently used.

3.5.4 The experience combing Dezyne and Nail

A problem was to connect the Nail code with the Dezyne code. At the start, we only compiled the Dezyne code. This was easy because of the provided Makefile. After this step, we tried to integrate the Nail code in the Dezyne code, but it was more challenging. The problem was in how we combine the `checksum.cc` file with the `btcp.nail.cc` file. After we solved this problem and adjusted our Makefile a little, the project compiled.

Chapter 4

Case study: TCP

This chapter describes the first case study of this paper. This case study looks at the implementation of TCP(see section 4.1) with the help of Dezyne and Nail. In sections 4.2 and 4.3, we describe the usage of both tools and the way how we implemented TCP in these tools. Section 4.4 gives an overview of how to put together the generated code of Dezyne and Nail. Finally, section 4.5 compares the implementation with the help of the tools and also describes the experience of working with tools practical.

4.1 TCP

At the beginning of this research, we planned to implement the security protocol EMV(see chapter 6). Nevertheless, we did not follow this early draft. After the first research to EMV, we knew that EMV is a completely new protocol to learn and to understand. This step would cost us a lot of time. Hence, we choose to implement the actual TCP for our second case study because bTCP and TCP do not differ a lot.

This case study only consists of the basic version of the TCP protocol. We only implement the classic connection opening and closing(see figures 4.1 and 4.2)). Also, we do not implement the simultaneous opening and closing part of the protocol. Further, this TCP implementation of us does not contain any congestion control algorithms like TCP Reno or TCP Tahoe. Another feature that is missing is the computation of the retransmission timer of TCP like it is described in the RFC 6298 [SCPA11].

Thereby, we split the case study into two parts. The first part is the server. The server handles incoming TCP connections and writes the transmitted data to his file system. The second part is the client. It tries to open a connection with the server and sends arbitrary data to the server. After sending the data, it will also close the connection with the server.

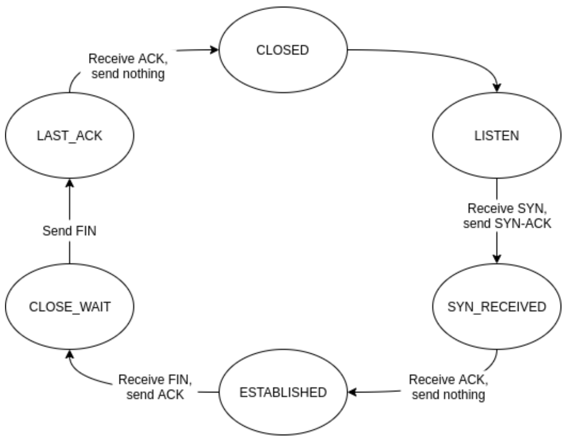


Figure 4.1: State machine of the a TCP server

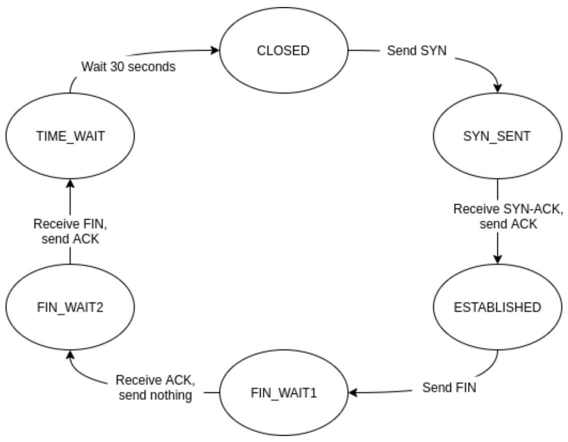


Figure 4.2: State machine of the a TCP client

As we can see in the figures 4.1 and 4.2, server and client share two states. The `CLOSED` is the beginning of a TCP connection. The server starts with a passive opening. The passive opening means that it starts to listen for an incoming connection. The client, however, starts with an active opening. Hence, the client actively sends an opening message to the server and starts the connection.

In the `ESTABLISHED` state, the server and the client exchange data with each other. Then the server or client, in our case the client, initializes the termination of the connection.

For starting and ending a connection, the server and client using a three-way handshake. You can find these handshakes in the figures 4.3 and 4.4.

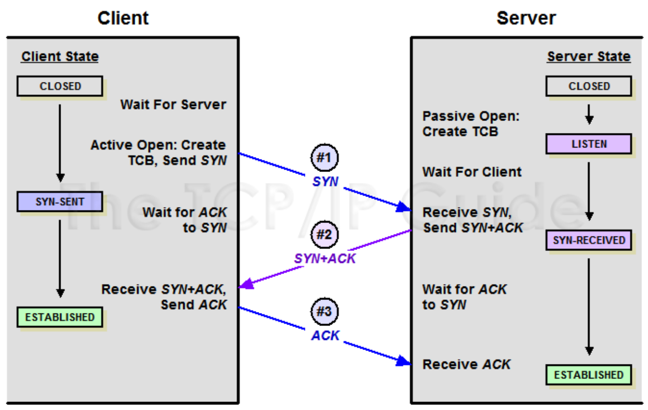


Figure 4.3: TCP opening procedure [TTG05b]

As already mentioned, the client starts the opening shake (figure 4.3) by sending an SYN packet. The packet contains a random initialize sequence number and a zero as an acknowledgment number. The server answers this packet with a SYN-ACK packet. For this packet, the server uses as sequence number a new number. As an acknowledgment number, it increments the received sequence number. After the client received the message, it answers with the ACK packet. The sequence number of this packet is equal to the acknowledgment number of the SYN-ACK packet. The acknowledgment number of the ACK packet is the increment sequence number of the SYN-ACK packet. After the server validates the packet the opening handshake is completed.

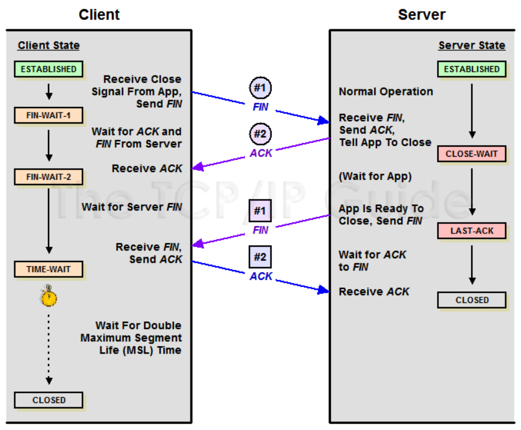


Figure 4.4: TCP closing procedure [TTG05a]

The closing handshake (figure 4.4) works almost in the same way as the opening handshake. The client starts with a FIN packet which indicates the start of the termination. The sequence number of this packet is arbitrary.

On this packet, the server answers with a packet whereby the sequence number is arbitrary and the acknowledgment number equals the sequence number of received packet increment by one. Additionally, the server sends a FIN packet itself with the same numbers. The client answers this packet with an ACK packet. The sequence number is now the acknowledgment number of the FIN packet of the server and the acknowledgment number is the sequence number but increment by one. After this step, the connection is terminated.

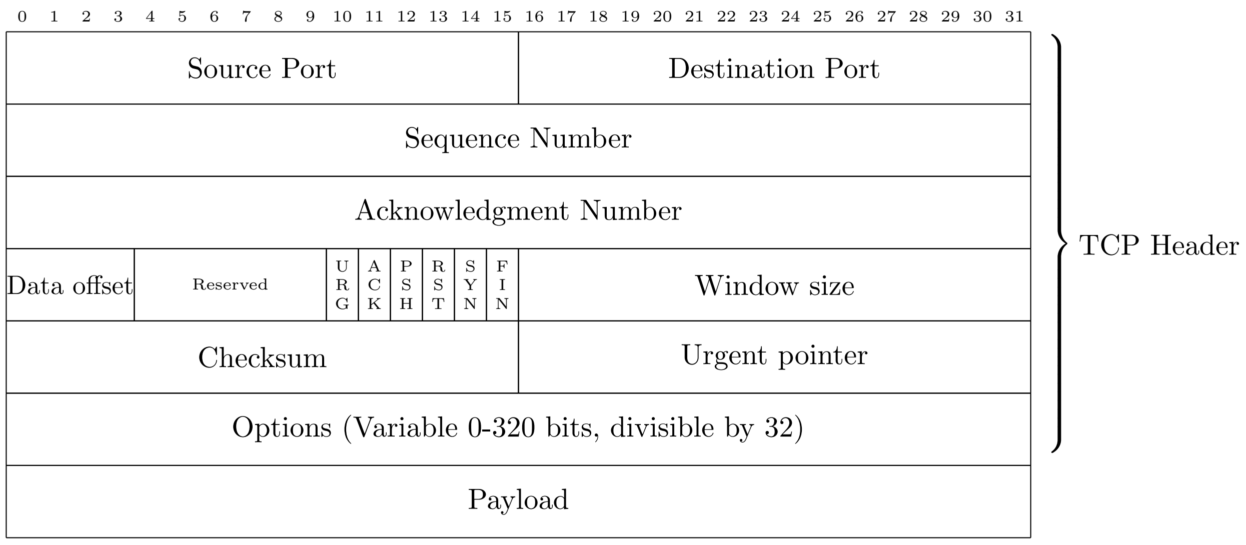


Figure 4.5: The TCP Header

The fields of the header (see figure 4.5) are defined the following way(based on the RFC793 [rfc81]):

1. Source Port(16 bits): The source port number.
2. Destination Port(16 bits): The destination port number.
3. Sequence number(32 bits): The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.
4. Acknowledgment number(32 bits): If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.
5. Data offset(4 bits): The number of 32-bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integer number of 32 bits long.

6. Reserved (6 bits): Reserved for future use.
7. Control Bits: 9 bits (from left to right):
   - URG: Urgent Pointer field significant
   - ACK: Acknowledgment field significant
   - PSH: Push Function
   - RST: Reset the connection
   - SYN: Synchronize sequence numbers
   - FIN: No more data from sender
8. Window size(16 bits): The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.
9. Checksum(16 bits): The 16-bit checksum field is used for error-checking of the header, the Payload and a Pseudo-Header. The Pseudo-Header consists of the Source IP Address, the Destination IP Address, the protocol number for the TCP-Protocol (0x0006) and the length of the TCP-Headers including Payload (in Bytes).
10. Urgent pointer(16 bits): This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.
11. Options(variable): Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary.
12. Payload: The data of the packet.

The header of a TCP packet (see figure 4.5)is different in comparison with the header of the bTCP packet (see figure 3.1 on page 11). The TCP header uses a source and destination port field to identify the origin of the packets. The sequence and acknowledgment number field are sorts of the same in both the headers. The difference in these fields is the size. The data offset field of TCP and the data length field of bTCP are completely different because the data offset field gives the size of the header and indicates the start of the sent data. The data length field, however, gives the size of the actual data. TCP defines the control flags in the separated fields. This takes more space than the control flags in the bTCP header. We need to compute the checksum in TCP in a different than in bTCP because in TCP we need to compute the checksum over pseudo-header(see above). The urgent pointer and options are new constructs of TCP. We defined this field in our case study for convenience, but we do not use these fields.

4.2 TCP in Dezyne

TCP in Dezyne In this section, we display the TCP client in the Dezyne modeling language. We chose to show the client of the TCP case study because we use some more functionalities of the language. You can find the client component in the several code fragments 4.1 and 4.2. For the full `Client` component take a look at our Gitlab repository[1]. This repository also contains the `Server` component with all its interface.

```
1  ...
2  component Client{
3      provides IClient iClient;
4      requires ISocketClient iSocketClient;
5      requires IFileHandler iFileHandler;
6
7      behaviour{
8
9          enum States {CLOSED, SYN_SENT, ESTABLISHED, FIN_WAIT1,
           ↪  FIN_WAIT2, TIME_WAIT};
10
11         States state = States.CLOSED;
12 ...
```

Code fragment 4.1: Component definition of TCP client

In lines 3-5, we define the interfaces which are required for the component and which are provided to the component. The `IClient` interface specifies the function of the client. For example, one function starts the connection to the server and has the name `startConnection`. This function already uses a new functionality that we did not use before. The function has an input parameter as any other function in a programming language.

The `Client` component provides the interface `IClient`. Therefore, we need to implement all the functions of the interface. As a help, we can use the functions of the interfaces `ISocketClient` and `IFileHandler`. The `ISocketClient` interface (see appendix B.2) provides the functions which we have to hand-write ourselves later on. The functions of this interface handle the packets with the help of the Nail code.

As the name already suggests the `IFileHandler` interface (see appendix B.3) handles the file that the client sends. The function `readFile` uses an extra functionality of the Dezyne modeling language. In this function, we use a `out` parameter. This parameter has the same meaning as a return value of the function. So, we use `readFile` to read a file and give this file to a function of `ISocketClient`.

[1] https://gitlab.science.ru.nl/pbongartz/bachelor-thesis/tree/master/TCP

The `IFileHandler` and `ISocketClient` have also the separate own components which implements the functions. These components can be found in the Gitlab repository.

Line nine of the code fragment 4.1 displays the states of the client. We copied these states from the state machine 4.2 on page 28. Finally, line 9 describes the initial sate of the client.

```
...
[state.ESTABLISHED]{
    on iClient.stopSending(): {
        ISocketClient.retval retval = iSocketClient.stop();
        if (ISocketClient.retval.retOK == retval){
            state = States.CLOSED;
            reply(IClient.retval.retOK);
        } else {
            reply(IClient.retval.retFailed);
        }
    }
    on iClient.sendData(): {
        IFileHandler.data file;
        IFileHandler.retval retFile =
        ↪  iFileHandler.readFile(file);
        if (retFile == IFileHandler.retval.retOK){
            ISocketClient.retval retSend =
            ↪  iSocketClient.sendData(file);
            if (ISocketClient.retval.retOK == retSend){
                state = States.ESTABLISHED;
                iFileHandler.stopReading();
                reply(IClient.retval.retOK);
            } else {
                reply(IClient.retval.retFailed);
            }
        }else{
            reply(IClient.retval.retFailed);
        }
    }
    on iClient.sendFin(): {
        ISocketClient.retval retval = iSocketClient.sendFin();
        if (ISocketClient.retval.retOK == retval){
            state = States.FIN_WAIT1;
            reply(IClient.retval.retOK);
        } else {
            reply(IClient.retval.retFailed);
        }
36        }
37    }
38    ...
```

Code fragment 4.2: Example implemantion of `behavior` block

Code fragment 4.2 shows an example implementation of the behavior block. In this code fragment, we take look in the state `ESTABLISHED`. In this state, we only allow three functions to execute. The first function `stopSending` just stops the whole `Client`. After executing this function, the `Client` goes back into its initial state and terminate the existing connection.

The interesting part of the `ESTABLISHED` state is the `sendData` function. As already mentioned, the `readFile` of the file handler interface has a return value. Therefore, we need to declare the variable `file` in line 13which has the data type `data`. The data type `data` is just type name of `vector`. `readFile` assigns data of file to the variable `file`. Now, we can use the variable for the `sendData` function of the `ISocketClient` interface which needs an input value. Further, the execution of this function do not trigger any state transion and the `Client` stays in the `ESTABLISHED`.

The `sendFin` only uses the function `sendFin` function of the `ISocketClient` inteface. After the function, the `Client` goes to the next state namely `FIN_WAIT1`.

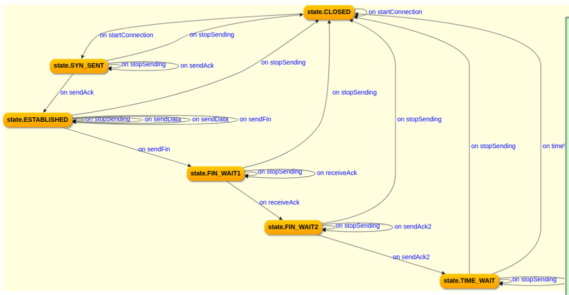


Figure 4.6: State machine of the TCP client

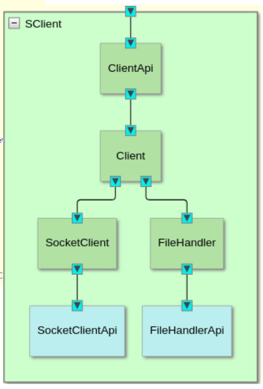


Figure 4.7: System overview of the TCP client

After modeling the component `Client` of the code fragment 4.2, Dezyne gives us the state machine in figure 4.6. This state machine is equal to the state machine 4.2 on page 28.

In figure 4.7, Dezyne shows us the complete client system. This system differs from the system(see figure 3.5 on page 16). The system contains several different components that we need to connect. The system is connected to the outer world with the component `ClientApi` which serves as a guard for illegal calls. Hence, we can only call the function of the `ClientApi` component. The `Client` has two required ports. The first port connects to the `SocketClient` component which is responsible for handling the sent and incoming packets of the client. The second port connects to the `FileHandler` component, which is responsible for reading the specified file.

The blue color of the components `SocketClientApi` and `SocketClient` indicates that these components need to be written by hand.

```
...
component SClient{

    provides IClientApi iClientApi;

    system{

        ClientApi clientApi;
        Client client;
        FileHandler fileHandler;
        SocketClient socketClient;
        SocketClientApi socketClientApi;
        FileHandlerApi fileHandlerApi;

        iClientApi <=> clientApi.iClientApi;
        clientApi.iClient <=> client.iClient;
        client.iFileHandler <=> fileHandler.iFileHandler;
        client.iSocketClient <=> socketClient.iSocketClient;

        socketClient.iSocketClientApi <=>
        ↪  socketClientApi.iSocketClientApi;
        fileHandler.iFileHandlerApi <=>
        ↪  fileHandlerApi.iFileHandlerApi;
    }
}
```

Code fragment 4.3: TCP client system in Dezyne

The code fragment 4.3 shows how we can obtain the figure 4.7. This code connects all components and interfaces. After programming the system, we are done with the modeling process in Dezyne.

The server of TCP has a lot of common with the server of the bTCP case study. Nevertheless, we changed the states of the TCP server to states in the state machine 4.1. The server uses some of the extra functionalities like the input value for functions. Also, the server has the same system structure as the client in figure 4.7. Besides the name changes, the server has not an extra component for handling data because it was not possible to implement this.

4.3 TCP in Nail

In figure 4.4, we can see the TCP header in Nail grammar format. We can already see that the 4.5 on page 30 do not differs a lot with header in the Nail grammar. The modeling process is almost a one-to-one translation. This makes the modeling easy.

Nevertheless, some difficulties occurred during the modeling process in the Nail grammar format. The difficulties occurred because of the differences between the header definitions on Wikipedia[2] and in the actual RFC [rfc81]. After some time, we decided to use the RFC as a reference that made the modeling process more clear.

Additionally, the header of TCP 4.4 and header of bTCP 3.3 have some difference. You can observe that the TCP header is more complex than the bTCP header.

```
options = {
    kind uint8
    length (optional uint8)
    value (optional uint16)
}

tcp = {
    source uint16
    destination uint16
    seq_num uint32
    ack_num uint32
    @data_offset uint4
    uint6 = 0
    urg uint1
    ack uint1
    psh uint1
    rst uint1
    syn uint1
    fin uint1
    window uint16
    @checksum uint16
    urg_pointer uint16
    transform checksum($current @checksum @data_offset)
```

[2]https://en.wikipedia.org/wiki/Transmission_Control_Protocol

```
24      $options, $body transform tcp_header($current
         ↪  @data_offset)
25      options apply $options (many options)
26      payload apply $body (many uint8)
27  }
```

Code fragment 4.4: TCP packet in Nail grammar

At the beginning of the TCP parser definition, we use the basic building blocks of the Nail grammar. The first fields of the TCP header are just simple values. For example, the sequence number of the TCP header is just a 32bits integer. We translate the sequence number into a 32-bit unsigned integer (see line 10). The first interesting field is the `data_offset` field(see line 12) because it displays the size of the header and so an important field for the function at the end of the TCP definition in the Nail grammar format. In line 13, we can see that 6-bits of the header are reserved for the future. These bits have to have the value zero. The flags are just one-bit fields. The checksum is also interesting because the field is also used self-written functions.

The first function (line 23) is the computation of the checksum. In this case study, we could implement the computation of the checksum because we need to compute the checksum over a pseudo-header that contains parts of the next higher protocol in the protocol stack: the IP protocol. With the set-up of our research, it is not possible to compute the checksum the right way. Therefore, we decided to ignore the checksum. However, the computation of such a checksum is not a problem.

The function `tcp_header` creates two new streams which we use later. The function gets as input the current stream and the data offset. With the help of the inputs, the generate function adds the option field and payload field to the bytes of the whole header and the parsing function creates the two new data object which we can use after the whole packet is parsed(see appendix B.4).

In the lines 25-26, we used new created data stream `$options` and `$payload` to define the last two fields of the TCP header. The `apply` syntax means that we apply the parser `options` to the stream `$options`. The type of options field is a new parser that is described in line 1-5. The payload field of the TCP header is just `many` 8-bit unsigned integers. `many` means that the field can contain of zero or more 8-bit integer.

4.4 Putting it together

Figure 4.8 describes the directory structure of our TCP project. We constructed the structure in the same ways as in our bTCP project (see figure

3.6). The folder `client_dzn` consists of the complete client system as we described it in section 4.2. Further, the file `tcp.nail` contains the Nail grammar format of the code fragment 4.4. For a full overview take a look at our Gitlab project.

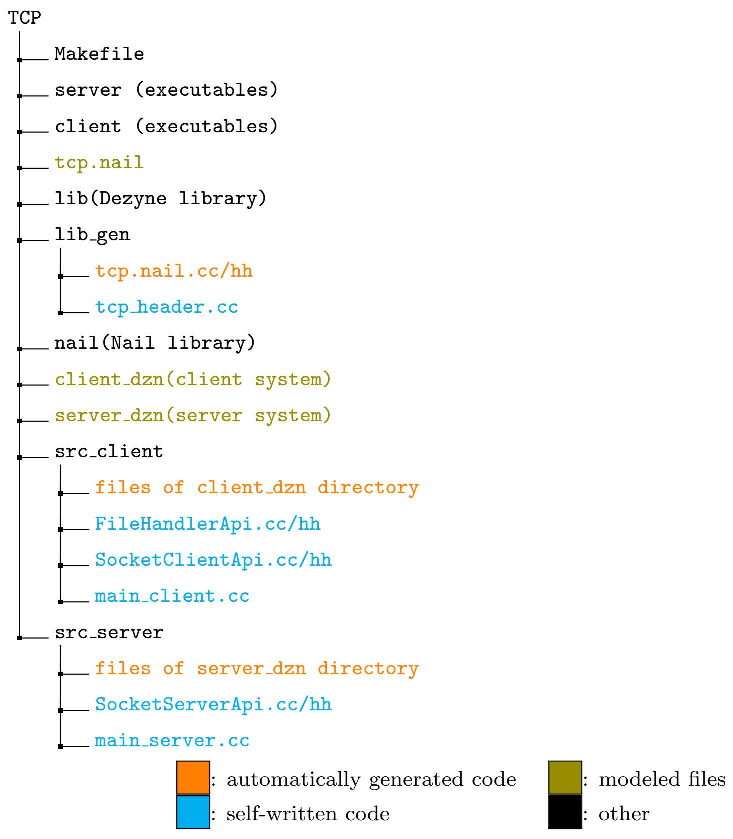


Figure 4.8: Directory structure of TCP

One difference with the bTCP project is the extra hand-written file in the `src_client` folder. The `FileHandlerApi` file corresponds to the change of modeling style (see section 4.2).

The file `tcp_header`(see appendix B.4) contains the functions signature of the checksum calculation, however, not the actual calculation as mentioned in section 4.3. Also, the file includes the function `tcp_header` which is necessary for the Nail code.

The Makefile of the TCP project looks almost the same as the Makefile A.3 of the bTCP project. Only some names changed to the TCP project.

```
#include <dzn/runtime.hh>
#include <dzn/locator.hh>

#include "SClient.hh"

```

```
int main(int argc, char* argv[]){
    dzn::locator loc;
    dzn::runtime rt;

    loc.set(rt);

    SClient sClient(loc);
    sClient.check_bindings();

    IClientApi::retval::type retStart = sClient.iClientApi.in.startConnection(5555);
    if(retStart == IClientApi::retval::retOK){
        IClientApi::retval::type retAck = sClient.iClientApi.in.sendAck();
        if (retAck == IClientApi::retval::retOK){
            IClientApi::retval::type retData = sClient.iClientApi.in.sendData();
            printf("data transfer completed\n");
            if (retData == IClientApi::retval::retOK){
                IClientApi::retval::type retFin = sClient.iClientApi.in.sendFin();
                if (retFin == IClientApi::retval::retOK){
                    IClientApi::retval::type retAck2 =
                    ↪  sClient.iClientApi.in.receiveAck();
                    if (retAck2 == IClientApi::retval::retOK){
                        IClientApi::retval::type retAck3 =
                        ↪  sClient.iClientApi.in.sendAck2();
                        if (retAck3 == IClientApi::retval::retOK){
                            IClientApi::retval::type retTime =
                            ↪  sClient.iClientApi.in.timeWait();
                            if (retTime == IClientApi::retval::retOK){
                                printf("protocol completed. connection closed.\n");
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Code fragment 4.5: main_client.cc file

The code fragment 4.5 looks more complex as the code fragment 3.4. But, both files do the same thing. They initialize the corresponding system and call the functions of the main component. In the case of the code fragment 4.5, we first call the function of the guard because the guard is connected with the outer-world (see section 4.2). The functions of the guard return value whether the function was executed successfully. In this case study, the client builds one connection with the server and then stops running.

```
...
//This function generates a new TCP packet
    size_t len;
    tcp packet;
    NailOutStream out;
    NailOutStream_init(&out, 4000);
    memset(&packet, 0, sizeof(packet));
    packet.source = dst;
    packet.destination = src;
    packet.seq_num = seq_num;
    packet.ack_num = ack_num;
    switch (flag_type) {
        case ACK:
```

```
        packet.ack = 1;;
        packet.urg = 0;
        packet.syn = 0;
        packet.psh = 0;
        packet.rst = 0;
        packet.fin = 0;
        break;
        case SYN_ACK:
        packet.ack = 1;
        packet.syn = 1;
        packet.urg = 0;
        packet.psh = 0;
        packet.rst = 0;
        packet.fin = 0;
        break;
        case FIN:
        packet.fin = 1;
        packet.ack = 0;
        packet.syn = 0;
        packet.urg = 0;
        packet.psh = 0;
        packet.rst = 0;
    }
    packet.window = window_size;
    packet.urg_pointer = urgent_pointer;
    packet.options.count = options_size;
    narray_alloc(packet.options, arena, packet.options.count);
    for (size_t i = 0; i < packet.options.count; i++) {
        options *o = &packet.options.elem[i];
        switch (options_type) {
            case 0:
            o->kind = 0;
            break;
            case 1:
            o->kind = 1;
            break;
            case 2:
            o->kind = 2;
            o->length = (uint8_t *) 4;
            o->value =(uint16_t *) 100;
            break;
        }
    }
    packet.payload.count = data_size;
    narray_alloc(packet.payload, arena, packet.payload.count);
    for (size_t i = 0; i < packet.payload.count; i++) {
        packet.payload.elem[i] = data[i];
    }

    //generates packet into arena(the actual usage of generate function of Nail code)
    if (gen_tcp(arena, &out, &packet) != 0) {
        printf("gen packet");
        return ISocketClientApi::retval::type::retFailed;
    }

    //gets the packet as bytes out of the arena
    const unsigned char *buf = NailOutStream_buffer(&out, &len)
...
```

Code fragment 4.6: Generate a TCP packet

The code fragment 4.6 shows how we can generate a new TCP packet.

With line 4, we create a new `tcp` struct. With the help of the `NailOutStream` object, we can extract the packet out of the `arena` where the packet is created. In the lines 8-38, we assign the new `tcp` object the necessary values. The flags are assigned by `switch`-block because we need to set several flags.

The lines 39-61 show the tricky parts of the TCP packet generation. We need to create `options` object which can be assigned to options field of the `tcp` object. We copied how we add data to the packet from the code fragment 3.5.

Line 64 calls the actual generating function of Nail and generates the packet in the `arena`. In line 70, we extract the packet from the `arena` and assign the packet to a variable. At this point, a packet is only a byte object and we can send it via socket to a client.

Parsing incoming data looks the same as in code fragment 3.6. Only we changed the function `parse_btcp` to `parse_tcp`. After parsing the byte object, we can use all values of `tcp` object.

4.5 Reflection

This section describes our experience using the tools Dezyne and Nail in the case study of TCP. Therefore, we give detailed experiences using Dezyne in section 4.5.1 and detailed experiences using Nail in section 4.5.2.

4.5.1 The experience using Dezyne

After the first case study, we took a deeper look at the documentation of Dezyne [dez]. On this web page, we found examples of systems that are used in the real world. These examples show the extra functionalities that we are describing in section 4.2. The features help with a better connection between two separate components.

As we started to use the feature of input and output variables of functions, our hand-written parts become more readable. A component had a clear purpose and it becomes easier to model this specific component.

Calling a function of guard component before actually calling the main component, makes the model more secure. The main component is defended by this guard. This makes clearer which functions engineers can call in certain states. Further, the guard stops the execution if it detects illegal behavior.

We could also use these functionalities in our first case study 3, too. Then the code would be less messy and the component had more distinct tasks.

We also experience how hard it can be to design a complex protocol like TCP in Dezyne. The basic transitions, which are shown in the figures 4.1 and 4.2 are easy to design. But, TCP has also exceptional behavior that is not easy to model. For example, we can name the simultaneously opening and closing sequences. The simultaneously closing sequence uses an extra state and therewith changes the model little bit.

After we modeled TCP in Dezyne, two questions came to our mind:

1. Does the Dezyne model correctly model TCP?
2. Is there a "better" way to model TCP in Dezyne?

In your case, we did not model the complete TCP protocol in Dezyne. In our model, some of the non-standard behavior is missing. We have still high confidence that we designed the best-scenario of TCP the right way.

The second question is hard to answer because we are not very experienced in the design protocol in Dezyne. To answer this question, we should talk with experienced users or the developers of Dezyne.

Last but not least, we think that the feature of verifying and simulating helped us a lot during the design process. It is easy to get lost in the model and to forget where we define some rules. Verifying helped us to find mistakes in the model and to fix them quickly.

4.5.2 The experience using Nail

Design the TCP header in the Nail grammar format does not differ with design the bTCP header. We only translate the fields of the TCP header in the Nail grammar format. This makes easy to model packets in Nail. With the experiences so far, we could model every packet structure. Even encapsulation like the network stack should be simple to design.

One problem was the ambiguous definition of the options fields in the RFC [rfc81] and other sources. The basic RFC only supports one kind of option. The other sources give a specification for other options, which made the design process harder. Hence, we focused on the RFC and implement the described options fields.

Another problem was to define the checksum computation of the TCP header. To calculate the checksum, we need fields of the next higher layer of the network stack. This is not possible in this case study because we do not know these fields. To compute the checksum, we could implement the full protocol stack and provide the checksum function with these fields.

4.5.3 The experience combing Dezyne and Nail

The biggest problem of this case study occurred as we want to connect the Dezyne and Nail code. The start was the same as in the case study 3. After programming the opening handshake of TCP, we wanted to send data via the packets(see code fragment 4.6). Generating the packet on the client does not raise any errors. After receiving the packet at the server, the parser returns an error and says the packet is invalid. We have an assumption of where the problem is. We think the problem lays in the initialize of the data field of the header. The field uses a `many` combinator (see line 26 of the code fragment 4.4). We did not find any example in the Nail project, which indicates the usage of this combinator. This makes it hard to fix this problem.

Chapter 5

Related Work

This chapter describes some of the related work of model-driven development and automatic code generation. Nowadays there are a lot of tools that help with automatic code generation and modeling a certain model. This chapter lists some of the other tools which we did not use in our research.

ComMa(Component Modeling and Analysis)

The tool ComMa(Component Modeling and Analysis)[coma][KSHS17] is a framework for model-based engineering by formalizing the interface specification. For the process of formalization, ComMa uses a new defined domain-specific language(DSL).

ComMa shares some of his functions with Dezyne. Nevertheless, ComMa also has some drawbacks in its functionality. One disadvantage of ComMa is that it is not possible to design a model like it is possible in Dezyne. Another drawback of ComMa is the fact that the tool does not support code generation. Besides this, ComMa cannot perform formal verification of the defined interfaces. However, Dezyne does not offer the possibility of time constraints and data constraints.

We cannot use ComMa in our case studies because the most important feature of Dezyne is missing in ComMa. It is not possible to generate code of the model.

Cogent

Cogent [ORC+16] is a purely functional programming language like Haskell. In these programming languages, programs are written as mathematical functions whereby the functions operate on algebraic data types. Nevertheless, Cogent differs from the most functional programming languages because Cogent is written for low-level operating system components [OCS+18]. Written Cogent code compiles into C code, and therefore it does not need any garbage collector, and memory management is explicit. Despite the explicit memory management, Cogent's uniqueness type system ensures memory safety.

The data description language and data refinement framework, Dargent [OCS+18], provides engineers with the possibility to define how Cogent describes its data types. It helps for a better transition to the generated C code. It would help to define data structure like packets, in higher-level languages like Cogent because of the more understandable representation of these languages.

Hammer parser generator

The Hammer [PH14] parser generator has a lot of similarities with Nail. Hammer introduces parsing on the level bits and bytes, and not only on the level of characters. Users of Hammer need to define the structure of data in several rules in the programming language. This restriction makes it harder for engineers to use the tool because the rules can be insecure and lead to security flaws of the parser. Further, Hammer can only parse data structures and cannot generate packets, which is a disadvantage towards Nail. Also, Hammer cannot handle input fields that have dependencies with each other. Hence, we cannot use Hammer as alternative of Nail.

Efficient Java Code Generation of Security Protocols Specified in *AnB* /*AnBx*

In 2014, Modesti [Mod14] presents a *AnBx* compiler and code generator. This tool generates Java code of security protocols. These protocols are noted in the simple Alice&Bob notation. This notation is a popular way to describe security protocols. But it is hard to create explicit checks that are needed for running security protocols based on an implicit concept like the *AnB* notation. Therefore, Modesti developed a compiler from the *AnB* notation to Java. He split this process into three parts. In the first part, the compiler translates the *AnBx* (an extension of *AnB*) protocol to *AnB* which is a suitable format for verification. In the second part, the compiler maps the *AnB* specifications into automatically generated code and optimize it. In the last part, the compiler finally generates Java code.

Chapter 6

Future Work

Alternative case study: SHH

One possible future research topic is a case study with the SHH protocol. SHH is standardized after some RFCs. These RFCs do not contain a state-machine of the protocol. This makes it harder for developers to program SHH. The research would look like this thesis. First, the researcher defines a state-machine for SHH. Secondly, he specifies this state-machine in Dezyne and defines the packet of SHH in Nail grammar. In the end, he has to combine both of the generated codes.

Alternative case study: EMV

A different research topic would be the implementation of the security protocol EMV. EMV is a payment method which is the technical standard for smart cards and payment terminals. This payment method is used by many companies that create credit cards, gift cards and debit cards like Visa and Mastercard. The standard is not only used in contact cards but also contactless cards and even mobile phones.

Investigate the compiler warnings and ToDo's of Nail code

Another research topic is to investigate the g++ warnings about the code and check whether an attacker can use the flaw that the warning points out to start an attack on the parser and generator. Further, the automatically generated code of Nail contains some ToDo's which suggest an overflow potential inside the Nail code. Additionally, the research can focus on the up-to-dateness of the Nail code and analyzes whether Nail can be improved.

Test robustness of generate code

A different research topic is to check the robustness of the generated code with a fuzz tester like AFL. With the help of such a fuzz tester, we can decide if Dezyne and Nail produce more secure code then hand-written code.

Implement complete TCP

In the case study in chapter 4, we implemented the best-case scenario of the TCP protocol. We could extend this implementation to the full version of TCP. Hence, we need to add the fallback mechanism of the protocol like different usage of flags and possibilities of simultaneous opening and closing. Further, we provide the usage of options, however, we do not use these options in our implementation. We programmed a very simple version of congestion control. We can extend the feature of congestion control to a more real-world version.

Compare our TCP implementation with real-world TCP implementations

The second case study 4 focuses on the extension of bTCP to TCP and the implementation. An extra step to this implementation would be to compare an existing implementation of TCP with our TCP implementation. In this comparison, we can take a look at the efficiency and the fastness of both the implementation.

Chapter 7

Conclusions

In this thesis, we implemented two stateful protocols, bTCP and TCP (see chapters 3 and 4) with the help of the tools Dezyne and Nail (see section 2.2). To evaluate these tools and see if we can combine them (see sections 3.4 and 4.4). Thereby, Dezyne is the tool for modeling the protocols state machines and creating the foundation of the source code of these protocols. In Nail grammar, we specify the structure of the packets of the respective protocols. Further, Nail creates the source code for parsing and generating these packets.

The two case studies are giving the following answers to our research question(see page 3): The most relevant answer is that it is achievable to program protocols with the help of Dezyne and Nail. Nevertheless, it is not the most straightforward thing to use these tools.

Dropping EMV for TCP

We already mentioned in section 4.1 that we had planned to implement the security protocol EMV(see section 6). After we performed the first case study, we started to research on EMV. The protocol is not hard to understand, nevertheless, we decided to drop the case study on EMV because of the time that we need to invest if we want to implement a new EMV implementation.

After this decision, it was obvious which stateful protocol we implement next. The decision fell on TCP because the bTCP protocol is just a smaller version of the real TCP. Hence, we could reuse some of the code of the first case study. This made us the second case study easier and we could focus on the modeling process of TCP in Dezyne(see section 4.2).

Experience using Dezyne

We already mention in the sections 2.2.1, 3.5.2 and 4.5.1 some of our experience using Dezyne.

Dezyne is a well-structured tool. As you can see in figure 2.1 on page 7,the stand-alone version of Dezyne is comparable to a modern IDE. The program offers syntax highlighting but also helps with syntax errors in the model. Furthermore, Vernum, the creators of Dezyne, update their tool on a regular base.

Furthermore, the process of verifying and simulating a model is easy to find in the tools. The results of the verification are simple to understand. As described in section 2.2.1, Dezyne can analyze the model based on five different checks. If an error occurs during the process of verification, Dezyne produces a traceback of the executed action and reports an error message what went wrong. We used this feature of Dezyne a lot during the modeling process. With this feature, engineers do not need to spend time checking their model on mistakes. They can easily use Dezyne itself to do these checks. For example, during the design process of TCP in Dezyne, we implement the behavior of the server. The behavior of the server is guarded by an extra component. We wanted to call a function in a state where the guard forbids this function. This mistake was easily discovered by the verifying feature.

The simulation of a model is also very simple to use. With only one click, Dezyne starts the simulation of the model and the user can click through possible functions in the model. For example, the verification feature of Dezyne finds a error and we can simulate the model to correct the error.

Besides the stand-alone version, Dezyne also provides a command-line tool that is used in both Makefiles (see code A.3 and B.5) of the project. Nevertheless, we can as well use the command-line tool in the modeling process.

Learning the Dezyne Modelling Language (DML) requires some time. DML is very complex and has a bundle of constructs that can be used to design the perfect model. After mastering DML, it becomes simple to design a new model in DML. Therefore, it saves time to design a protocol.

After the modeling process in both case studies, we had two remaining questions:

1. Does the Dezyne model correctly model the protocol?
2. Is there a ”better” way to model the protocol in Dezyne?

We can answer the first question looking at the state machines which are provided by the protocols RFC or different sources. For the second question, we need to talk with the developers of Dezyne or experience users which can help us build better models in Dezyne.

All in all, we can conclude that Dezyne helps develop a secure protocol. With the usage of Dezyne, a developer needs to think before a starts programming because he first needs to specify the state diagram of the protocol. Additionally, the tool makes the developing process more structured because programmers always have an overview of what he is currently programming. Hence, programmers are more confident and save time. Nevertheless, a developer needs to learn the DML of Dezyne and how to translate the protocol from an RFC to DML.

Experience using Nail

We already mention in the sections 2.2.2, 3.5.3 and 4.5.2 some of our experience using Nail.

One of the experiences is that Nail is not well documented. Therefore, it is hard to use the tool at the beginning. After getting to know the Nail grammar, it becomes simple to define a new structure of a packet.

Nevertheless, a programmer needs to program some extra lines of code to use the automatically generated code of Nail. The tool uses so-called type `NailArena` to parse and generate packages[BZ14a]. So, a programmer needs keeping this in his mind if he programs with generated Nail code.

A remarkable fact of Nail is how many lines of code the tool generates from a little input specification. The grammar 3.3 has only 15 lines of code. In comparison, the generated code of this grammar has roughly 428 lines of code. Hence, the grammar is very lightweight. Even after some settling-in period, the tool in good integrated into the programming flow.

The tool was a project of a Master thesis and is already four years old. Therefore, it is not validated if Nail is nowadays security-relevant. If we compiled Nail code with Wall flag of the g++ compiler, we got some warnings (see section 3.5.3). Besides these warnings, the automatically generate code of Nail contains some ToDo's which need correction. These problems can be solved in the future (See chapter 6).

In conclusion, we can say that Nail is a really helpful tool. It saves time programming a protocol because a programmer does not need to figure out how to parse or generator the packets of the protocol. Thereby, Nail has some security methods in place. These methods are preventing attacks like shell-code in padding fields, which are corrupting the memory (as discussed in [BZ14a]). After some setting-in period, the usage of Nail felt very natural and easy.

Experience putting Dezyne and Nail code together

The third part of the two case studies was to put the generated code of Dezyne and Nail together. The experience of the individual case studies can be found in the sections 3.4 and 4.4.

In the beginning, it was not obvious to combine the code of Dezyne and Nail. We started with understanding and using the automated generated code of Dezyne.

The Dezyne code needs several elements before it runs. For example, the modeled system needes a `dzn::locator` and a `dzn::runtime`(see sections 3.4 and 4.4). These parts are crucial for running the Dezyne code. Hence, software engineers need to know these parts exist and they need to implement the parts before starting something else.

Another point of the Dezyne code is the hand-written parts. For example, in our case studies, the creation of sockets, getting, parsing and sending packets(The parsing part is handled by the Nail code). It is important to know that engineers can only influence the code of these parts of the whole system. It is also possible to manipulate the generated code, but this would destroy the whole system and we did not achieve anything.

After we got the Dezyne code running, we started to combine the Dezyne code with the Nail code. As we already mentioned in sections 3.3, 3.4, 3.5.3, 4.5.2, 4.4 and 4.5.2, engineers have to program some parts of the parser and generator themselves. For example, in both case studies, we needed to implement the checksum function ourselves (see sections 3.4 and 4.4). Hence, the generated code of Nail gives us the needed parts of the packet to compute the checksum. Therefore, Nail needs one method for parsing the checksum and one method for generating the checksum (see code fragments A.2 and B.4). These methods need some basic knowledge of how bytes are working because the parts in these methods are bytes.

In conclusion, we can say that the combing code of Dezyne and Nail together is at the begin not trivial. After some time to settle in and understanding how the code works, it becomes very clear how to use the code of both tools. The Dezyne code is the framework in which we use the parse and generate functions of the Nail.

Conclusions

All in all, we can answer after the two case studies the following three questions:

1. Is it possible to use Dezyne and Nail for automatic code generation of protocols?

2. Is it easy to use Dezyne and Nail during the implementation of protocols?

3. Is it better to use Dezyne and Nail during the development of protocols?

As we already mentioned in the sections above, Dezyne and Nail are expressive enough to model protocols like TCP. In Dezyne, developers have many features to model stateful protocols like TCP. The usage of Nail almost feels like a one-to-one translation of the specifications to the Nail grammar format. Nevertheless, Nail also has some flaws. In the case study 4, we could not send data via the generated packet of Nail.

The second question is not trivial compared to the first question. Both tools were new for use at the beginning. Hence, it took us some time to understand the feature of both tools. For example, Dezyne uses a new modeling language that we need to learn. Further, the Nail grammar format was not easy to understand at the beginning. After some time, we understand both the Dezyne modeling language and the Nail grammar format. This made the usage of Dezyne and Nail easier and also helped during the implementation of both case studies.

The third question is hard to answer because it is not clear what better means in this context. Hence, we think about the maintainability of a project that is developed with the help of Dezyne and Nail. As engineers encounter a mistake in the flow of the protocol during the testing phase. They do not need to change parts of the source code because they make changes on a higher level namely in Dezyne. Further, this also holds for Nail. If engineers want to change the structure of the packet, they can do it in the Nail grammar of the packet.

We can answer the third question in the context of whether the usage of Dezyne and Nail lowers the likeliness of a bug in the project. With Dezyne it is less likely to have a bug in the project because Dezyne always verifies and validates the model before it generates the code of the model. Hence, engineers can catch errors in the design phase and not in the testing phase. Nail eliminates the risk of parsing bytes packets by hand-written code. Nevertheless, the Nail code produces g++ warnings which can cause bugs and create attack vectors for an attacker.

Bibliography

[APS14] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, Jan 2014.

[BHH$^+$17] S. Bratus, L. Hermerschmidt, S.M. Hallberg, M. Locasto, F.D. Momot, M.L. Patterson, and A Shubina. Curing the vulnerable parser: Design patterns for secure input handling. *;login: The USENIX Magazine*, Vol. 42(1):32–39, Spring 2017.

[BZ14a] J. Bangert and N. Zeldovich. Nail: A practical interface generator for data formats. In *2014 IEEE Security and Privacy Workshops*, pages 158–166, May 2014.

[BZ14b] J. Bangert and N. Zeldovich. Nail: A practical interface generator for data formats. In *Login Journal*, 2014.

[BZ15a] J. Bangert and N. Zeldovich. Nail - a practical tool for parsing and generating data formats. Master's thesis, Massachusetts Institute of Technology, 2015.

[BZ15b] Julian Bangert and Nickolai Zeldovich. Nail. `https://github.com/jbangert/nail`, 2015.

[coma] ComMa. `http://comma.esi.nl/`. Accessed: 2019-11-02.

[comb] Compiler-compiler. `https://en.wikipedia.org/wiki/Compiler-compiler`. Accessed: 2019-11-05.

[dez] Dezyne description from Verum. `https://www.verum.com/dezyne/`. Accessed: 26-09-2019.

[KLM15] John Klein, Harry Levinson, and Jay Marchetti. Model-driven engineering: Automatic code generation and beyond. Technical Report CMU/SEI-2015-TN-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2015.

[KSHS17] Ivan Kurtev, Mathijs Schuts, Jozef Hooman, and Dirk-Jan Swagerman. Integrating interface modeling and analysis in an industrial setting. In *MODELSWARD*, 2017.

[Mod14] Paolo Modesti. Efficient Java Code Generation of Security Protocols Specified in AnB/AnBx. In Sjouke Mauw and Christian Damsgaard Jensen, editors, *Security and Trust Management*, pages 204–208, Cham, 2014. Springer International Publishing.

[OCS$^+$18] Liam O'Connor, Zilin Chen, Partha Susarla, Christine Rizkallah, Gerwin Klein, and Gabriele Keller. Bringing effortless refinement of data layouts to cogent. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, pages 134–149, Cham, 2018. Springer International Publishing.

[ORC$^+$16] Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby Murray, and Gerwin Klein. Cogent: Certified compilation for a functional systems language, 2016.

[PH14] Meredith Patterson and Dan Hirsch. Hammer parser generator. https://github.com/UpstandingHackers/hammer, 2014.

[rfc81] Transmission Control Protocol. RFC 793, September 1981.

[SCPA11] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

[Sel03] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, Sep. 2003.

[TTG05a] A TCP/IP reference You can Understand! The TCP/IP Guide. Tcp connection termination procedure. http://www.tcpipguide.com/free/t_TCPConnectionTermination-2.htm, 2005. [Online; accessed December 13, 2019].

[TTG05b] A TCP/IP reference You can Understand! The TCP/IP Guide. Tcp sequence number synchronization. http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentSequenceNumberSynchroniz-2.htm, 2005. [Online; accessed December 13, 2019].

[Wik19] Wikipedia. Modellgetriebene Softwareentwicklung — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=Modellgetriebene_Softwareentwicklung&oldid=188569540, 2019. [Online; accessed 7-January-2020].

Appendix A

Appendix: Case study 1

This appendix contains some extra file of our first case study(see chapter 3). The structure of the appendix is equal to the structure for chapter 3. The whole project can be found on Gitlab[1].

A.1 bTCP in Dezyne

A.1.1 IUtilServer.dzn

```
interface IUtilServer {
    in void wait_for_syn();
    in void send_syn_ack();
    in void receive_data();
    in void wait_for_fin();
    in void receive_final_ack();

    behaviour{
        on wait_for_syn: {}
        on send_syn_ack: {}
        on receive_data: {}
        on wait_for_fin: {}
        on receive_final_ack: {}
    }
}

component UtilServer{
    provides IUtilServer iUtilServer;
}
```

Code fragment A.1: IUtilServer.dzn

A.2 bTCP in Nail

A.2.1 checksum.cc

```
#include "btcp.nail.hh"
#include "memstream.hh"
#include <arpa/inet.h>
```

[1]https://gitlab.science.ru.nl/pbongartz/bachelor-thesis/tree/master/bTCP

```
#include <zlib.h>

#define POLY 0xedb88320


template<typename str>
struct checksum_parse {
};


template <> struct checksum_parse<NailMemStream>{
    static int f(NailArena *tmp, NailMemStream *header, NailMemStream *payload,
    ↪  uint32_t *ptr_checksum){
        const uint32_t Polynomial = 0xEDB88320;
        size_t length1 = header->getSize();
        size_t length2 = payload->getSize();
        uint32_t previousCrc32 = 0;
        unsigned char* field1 = (unsigned char*) header->getBuf();
        unsigned char* field2 = (unsigned char*) payload->getBuf();

        uint32_t crc = ~previousCrc32;

        while (length1--)
        {
            crc ^= *field1++;
            for (unsigned int j = 0; j < 8; j++)
                if (crc & 1)
                    crc = (crc >> 1) ^ Polynomial;
                else
                    crc =  crc >> 1;
        }

        while (length2--)
        {
            crc ^= *field2++;
            for (unsigned int j = 0; j < 8; j++)
                if (crc & 1)
                    crc = (crc >> 1) ^ Polynomial;
                else
                    crc =  crc >> 1;
        }

        if (~crc == *ptr_checksum){
            return 0;
        } else{
            return -1;
        }
    }
};

template<typename str>
struct crc32_checksum_parse {
};


template <> struct crc32_checksum_parse<NailMemStream>{
    static int f(NailArena *tmp, NailMemStream *in_current,uint32_t *ptr_checksum){
        NailMemStream header(in_current->getBuf(), (size_t) 10);
        NailMemStream payload(in_current->getBuf()+16, (size_t) 1000);
        return checksum_parse<NailMemStream>::f(tmp, &header, &payload,
        ↪  ptr_checksum);
    }
};



int checksum_generate(NailArena *tmp, NailOutStream *header, NailOutStream *payload,
↪ uint32_t *ptr_checksum){
    const uint32_t Polynomial = 0xEDB88320;
    size_t length1 = header->size;
    size_t length2 = payload->size;
    uint32_t previousCrc32 = 0;
    unsigned char* field1 = (unsigned char*) header->data;
    unsigned char* field2 = (unsigned char*) payload->data;

    uint32_t crc = ~previousCrc32;

    while (length1--)
    {
        crc ^= *field1++;
        for (unsigned int j = 0; j < 8; j++)
            if (crc & 1)
                crc = (crc >> 1) ^ Polynomial;
            else
                crc =  crc >> 1;
    }

    while (length2--)
    {
        crc ^= *field2++;
        for (unsigned int j = 0; j < 8; j++)
            if (crc & 1)
                crc = (crc >> 1) ^ Polynomial;
            else
                crc =  crc >> 1;
    }

    *ptr_checksum = ~crc;
    return 0;
}

int crc32_checksum_generate(NailArena *tmp, NailOutStream *current, uint32_t
↪ *ptr_checksum){
    NailOutStream header = {
            .data =current->data,
            .size = (size_t) 10,
            .pos = (size_t) 10,
            .bit_offset = 0
    };

    NailOutStream payload = {
            .data =current->data+16,
            .size = (size_t) 1000,
            .pos = (size_t) 1000,
            .bit_offset = 0
    };
    return checksum_generate(tmp, &header, &payload, ptr_checksum);
}



#include "btcp.nail.cc"
```

Code fragment A.2: checksum.cc

A.3 Putting it together

A.3.1 Makefile

```
CXX = g++
CXXFLAGS = -std=gnu++11 -I$(RUNTIME) -I$(SRC_SERVER) -I$(SRC_CLIENT) -I$(NAIL_LIB)
↪  -I$(LIB) -Wall -lrt
CPPFLAGS = -I$(SRC) -I$(RUNTIME)
LDFLAGS = -lpthread -lz
RM = rm -rf
VERSION = 2.9.1
CURRENT := $(shell dzn query | grep '*' | sed 's,\* ,,')

ifneq ($(VERSION),$(CURRENT))
$(info current version: $(CURRENT) is not equal to selected version: $(VERSION))
endif

SRC_SERVER = ./src_server
SRC_CLIENT = ./src_client
RUNTIME = ./lib
GEN = ./lib_gen
NAIL_LIB = ./nail
LIB =  /usr/include

SRCS_SERVER = $(filter-out $(SRC_SERVER)/btcp.nail.cc, $(wildcard
↪  $(SRC_SERVER)/*.cc))
SRCS_SERVER += $(wildcard $(RUNTIME)/*.cc)

SRCS_CLIENT = $(filter-out $(SRC_CLIENT)/btcp.nail.cc, $(wildcard
↪  $(SRC_CLIENT)/*.cc))
SRCS_CLIENT += $(wildcard $(RUNTIME)/*.cc)


OBJS_SERVER = $(subst .cc,.o,$(SRCS_SERVER))
OBJS_CLIENT = $(subst .cc,.o,$(SRCS_CLIENT))

NAIL = $(wildcard *.nail)

all:
make server client

server: $(OBJS_SERVER)
$(CXX) $(CXXFLAGS) -o server $(OBJS_SERVER) $(LDFLAGS)

client: $(OBJS_CLIENT)
$(CXX) $(CXXFLAGS) -o client $(OBJS_CLIENT) $(LDFLAGS)

runtime: | $(RUNTIME)/dzn
for f in $(shell dzn ls -R /share/runtime/c++ | sed 's,/c++/,,' | tail -n +3); do \
dzn cat --version=$(VERSION) /share/runtime/c++/$$f > $(RUNTIME)/$$f; \
done
touch $@

$(RUNTIME)/dzn:
mkdir -p $@

generate_all: generate_client generate_server generate_parser

generate_client: $(wildcard client_dzn/*.dzn)
```

```
for f in $^; do dzn -v code --version=$(VERSION) -l c++ -o $(SRC_CLIENT) $$f; done
touch $@

generate_server: $(wildcard server\_dzn/*.dzn)
for f in $^; do dzn -v code --version=$(VERSION) -l c++ -o $(SRC_SERVER) $$f; done
touch $@

generate_parser: ; ~/Projekte/nail/generator/nail $(NAIL) ; cp btcp.nail.hh
↪  btcp.nail.cc $(GEN)

clean:
$(RM) $(OBJS_CLIENT) $(OBJS_SERVER) server client

clean_generated:
$(RM) `grep -h dzn $(SRC)/*.dzn.d | sed -e 's,:.*,,' -e 's,%,.,g'`
$(RM) $(RUNTIME)
$(RM) runtime generate

distclean: clean clean_generated
$(RM) *~
```

Code fragment A.3: Makefile of the bTCP project

Appendix B

Appendix: Case study 2

This appendix contains some extra file of our first case study(see chapter 4). The structure of the appendix is equal to the structure for chapter 4. The whole project can be found on Gitlab[1].

B.1 TCP in Dezyne

B.1.1 IClient.dzn

```
interface IClient{

    extern xint $int$;

    enum retval { retOK, retFailed};

    in retval startConnection(in xint port);
    in retval stopSending();
    in retval sendAck();
    in retval sendAck2();
    in retval sendFin();
    in retval receiveAck();
    in retval timeWait();
    in retval sendData();

    behaviour{

        enum States {CLOSED, SYN_SENT, ESTABLISHED, FIN_WAIT1, FIN_WAIT2,
        ↪  TIME_WAIT};

        States state = States.CLOSED;

        [state.CLOSED]{
            on stopSending: illegal;
            on sendAck: illegal;
            on sendAck2: illegal;
            on sendData: illegal;
            on receiveAck: illegal;
            on timeWait: illegal;
            on sendFin: illegal;
            on startConnection: {
                reply(retval.retFailed);
            }
```

[1]https://gitlab.science.ru.nl/pbongartz/bachelor-thesis/tree/master/bTCP

```
            on startConnection: {
                state = States.SYN_SENT;
                reply(retval.retOK);
            }
        }
        [state.SYN_SENT]{
            on startConnection: illegal;
            on sendData: illegal;
            on receiveAck: illegal;
            on timeWait: illegal;
            on sendFin: illegal;
            on sendAck2: illegal;
            on stopSending: {
                reply(retval.retFailed);
            }
            on stopSending: {
                state = States.CLOSED;
                reply(retval.retOK);
            }
            on sendAck: {
                reply(retval.retFailed);
            }
            on sendAck: {
                state = States.ESTABLISHED;
                reply(retval.retOK);
            }
        }
        [state.ESTABLISHED]{
            on startConnection: illegal;
            on sendAck: illegal;
            on receiveAck: illegal;
            on timeWait: illegal;
            on sendAck2: illegal;
            on stopSending: {
                reply(retval.retFailed);
            }
            on stopSending: {
                state = States.CLOSED;
                reply(retval.retOK);
            }
            on sendData: {
                reply(retval.retOK);
            }
            on sendData: {
                reply(retval.retFailed);
            }
            on sendFin: {
                state = States.FIN_WAIT1;
                reply(retval.retOK);
            }
            on sendFin: {
                reply(retval.retFailed);
            }
        }
        [state.FIN_WAIT1]{
            on startConnection: illegal;
            on sendAck: illegal;
            on sendData: illegal;
            on timeWait: illegal;
            on sendFin: illegal;
            on sendAck2: illegal;
            on stopSending: {
                reply(retval.retFailed);
            }
            on stopSending: {
                state = States.CLOSED;
                reply(retval.retOK);
            }
            on receiveAck: {
                reply(retval.retFailed);
            }
            on receiveAck: {
                state = States.FIN_WAIT2;
                reply(retval.retOK);
            }
        }
        [state.FIN_WAIT2]{
            on startConnection: illegal;
            on receiveAck: illegal;
            on sendData: illegal;
            on timeWait: illegal;
            on sendFin: illegal;
            on sendAck: illegal;
            on stopSending: {
                reply(retval.retFailed);
            }
            on stopSending: {
                state = States.CLOSED;
                reply(retval.retOK);
            }
            on sendAck2: {
                reply(retval.retFailed);
            }
            on sendAck2: {
                state = States.TIME_WAIT;
                reply(retval.retOK);
            }
        }
        [state.TIME_WAIT]{
            on startConnection: illegal;
            on receiveAck: illegal;
            on sendData: illegal;
            on sendFin: illegal;
            on sendAck: illegal;
            on sendAck2: illegal;
            on stopSending: {
                reply(retval.retFailed);
            }
            on stopSending: {
                state = States.CLOSED;
                reply(retval.retOK);
            }
            on timeWait: {
                reply(retval.retFailed);
            }
            on timeWait: {
                state = States.CLOSED;
                reply(retval.retOK);
            }
        }
    }
}
```

Code fragment B.1: Interface for client in Dezyne

```
interface ISocketClient
{
    extern data $std::vector<unsigned char>$;
    extern xint $int$;
    enum retval { retOK, retFailed };

    in retval start(in xint port);
    in retval stop();
    in retval sendAck();
    in retval sendAck2();
    in retval sendFin();
    in retval sendData(in data file);
    in retval receiveAck();
    in retval timeWait();

    behaviour {
        enum State { Off, Waiting, Connected };

        State state = State.Off;

        [state.Off]{
            on stop: illegal;
            on sendAck: illegal;
            on sendAck2: illegal;
            on sendFin: illegal;
            on sendData: illegal;
            on receiveAck: illegal;
            on timeWait: illegal;
            on start: {
                state = State.Waiting;
                reply(retval.retOK);
            }
            on start: {
                reply(retval.retFailed);
            }
        }
        [state.Waiting] {
            on start: illegal;
            on receiveAck: illegal;
            on sendFin: illegal;
            on sendData: illegal;
            on sendAck2: illegal;
            on timeWait: illegal;
            on stop: {
                state = State.Off;
                reply(retval.retOK);
            }
            on stop: {
                reply(retval.retFailed);
            }
            on sendAck: {
                state = State.Connected;
                reply(retval.retOK);
            }
            on sendAck: {
                reply(retval.retFailed);
            }
        }
        [state.Connected]{
            on start: illegal;
            on sendAck: illegal;
            on sendAck2: {
                state = State.Connected;
                reply(retval.retOK);
            }
            on sendAck2: {
                reply(retval.retFailed);
            }
            on stop: {
                state = State.Off;
                reply(retval.retOK);
            }
            on stop: {
                reply(retval.retFailed);
            }
            on receiveAck:{
                reply(retval.retOK);
            }
            on receiveAck:{
                reply(retval.retFailed);
            }
            on timeWait: {
                state = State.Off;
                reply(retval.retOK);
            }
            on timeWait: {
                reply(retval.retFailed);
            }
            on receiveAck: {
                reply(retval.retOK);
            }
            on receiveAck: {
                reply(retval.retFailed);
            }
            on sendData: {
                reply(retval.retOK);
            }
            on sendData: {
                reply(retval.retFailed);
            }
            on sendFin: {
                reply(retval.retOK);
            }
            on sendFin: {
                reply(retval.retFailed);
            }
        }
    }
}
```

Code fragment B.2: Interface of socket for client in Dezyne

```
interface IFileHandler
{
    extern data $std::vector<unsigned char>$;
    enum retval { retOK, retFailed };

    in retval readFile(out data file);
    in void stopReading();

    behaviour {
            on readFile: {reply (retval.retOK);}
            on readFile: {reply (retval.retFailed);}
            on stopReading: {}
    }
}
```

Code fragment B.3: Interface of file handler for client in Dezyne

B.2 TCP in Nail

B.2.1 tcp_header.cc

```
#include "tcp.nail.hh"
#include "memstream.hh"
#include "size.hh"

template <typename str> struct checksum_parse{
    static int f(NailArena *tmp, NailMemStream *in_current, uint16_t *ptr_checksum,
    ↪ uint8_t *ptr_offset){
        return 0;
    }
};

int checksum_generate(NailArena *tmp, NailOutStream *current, uint16_t
↪ *ptr_checksum, uint8_t *ptr_offset){
    return 0;
}

template <typename str> struct tcp_header_parse{
    typedef NailMemStream out_1_t;
    typedef NailMemStream out_2_t;

    static int f(NailArena *tmp, NailMemStream **out_options, NailMemStream
    ↪ **out_body, NailMemStream *in_current,  uint8_t *ptr_offset){
        if (n_fail(in_current->repositionOffset(*ptr_offset*4, 0))) return -1;
        *out_options = (NailMemStream *) n_malloc(tmp, sizeof(NailMemStream));
        new((void *) *out_options) NailMemStream(in_current->getBuf()+20,
        ↪ (*ptr_offset*4)-20);
        *out_body = (NailMemStream *) n_malloc(tmp, sizeof(NailMemStream));
        new((void *) *out_body) NailMemStream(in_current->getBuf()+*ptr_offset*4,
        ↪ in_current->getSize()-(*ptr_offset*4));
        return 0;
    }
};

int tcp_header_generate(NailArena *tmp, const NailOutStream *in_options, const
↪ NailOutStream *in_body, NailOutStream *current, uint8_t *ptr_offset){
    if (in_options->pos % 3 != 0) return -1;
    if (in_options->pos > 40) return  -1;

    if (n_fail(size_generate(tmp, in_options, current, ptr_offset))) return -1;
    *ptr_offset = 5 + (in_options->pos % 3);
    if(n_fail(tail_generate(tmp, in_body, current))) return -1;
    return 0;
}

#include "tcp.nail.cc"
```

Code fragment B.4: tcp_header.cc

B.3 Putting it together

B.3.1 Makefile

```
CXX = g++
CXXFLAGS = -std=gnu++11 -I$(RUNTIME) -I$(SRC_SERVER) -I$(SRC_CLIENT) -I$(NAIL_LIB)
↪  -I$(LIB) -Wall -lrt -g
CPPFLAGS = -I$(SRC) -I$(RUNTIME)
LDFLAGS = -lpthread -lz
RM = rm -rf
VERSION = 2.9.1
CURRENT := $(shell dzn query | grep '*' | sed 's,\* ,,')

ifneq ($(VERSION),$(CURRENT))
$(info current version: $(CURRENT) is not equal to selected version: $(VERSION))
endif

SRC_SERVER = ./src_server
SRC_CLIENT = ./src_client
RUNTIME = ./lib
GEN = ./lib_gen
NAIL_LIB = ./nail
LIB =  /usr/include

SRCS_SERVER = $(filter-out $(SRC_SERVER)/tcp.nail.cc, $(wildcard
↪  $(SRC_SERVER)/*.cc))
SRCS_SERVER += $(wildcard $(RUNTIME)/*.cc)

SRCS_CLIENT = $(filter-out $(SRC_CLIENT)/tcp.nail.cc, $(wildcard
↪  $(SRC_CLIENT)/*.cc))
SRCS_CLIENT += $(wildcard $(RUNTIME)/*.cc)


OBJS_SERVER = $(subst .cc,.o,$(SRCS_SERVER))
OBJS_CLIENT = $(subst .cc,.o,$(SRCS_CLIENT))

NAIL = $(wildcard *.nail)

all:
make server client

server: $(OBJS_SERVER)
$(CXX) $(CXXFLAGS) -o server $(OBJS_SERVER) $(LDFLAGS)

client: $(OBJS_CLIENT)
$(CXX) $(CXXFLAGS) -o client $(OBJS_CLIENT) $(LDFLAGS)

runtime: | $(RUNTIME)/dzn
for f in $(shell dzn ls -R /share/runtime/c++ | sed 's,/c++/,,' | tail -n +3); do \
dzn cat --version=$(VERSION) /share/runtime/c++/$$f > $(RUNTIME)/$$f; \
done
touch $@

$(RUNTIME)/dzn:
mkdir -p $@

generate_all: generate_client generate_server generate_parser

generate_client: $(wildcard client_dzn/*.dzn)
```

```
for f in $^; do dzn -v code --version=$(VERSION) -l c++ -o $(SRC_CLIENT) $$f; done
touch $@

generate_server: $(wildcard server_dzn/*.dzn)
for f in $^; do dzn -v code --version=$(VERSION) -l c++ -o $(SRC_SERVER) $$f; done
touch $@

generate_parser: ; ~/Projekte/nail/generator/nail $(NAIL) ; cp tcp.nail.hh
↪  tcp.nail.cc $(GEN) ; rm tcp.nail.hh tcp.nail.cc

clean:
$(RM) $(OBJS_CLIENT) $(OBJS_SERVER) server client

clean_generated:
$(RM) `grep -h dzn $(SRC)/*.dzn.d | sed -e 's,:.*,,' -e 's,%,.,g'`
$(RM) $(RUNTIME)
$(RM) runtime generate

distclean: clean clean_generated
$(RM) *~
```

Code fragment B.5: Makefile of the bTCP project