MASTER'S THESIS

Abstractions for software communication

Programming protocols in typescript

van Overveld, J. (Jan)

Award date:
2020

ABSTRACTIONS FOR SOFTWARE COMMUNICATION

PROGRAMMING PROTOCOLS IN TYPESCRIPT

THESIS

Author	Jan van Overveld
Student number	
Date of defence	7 February 2020

ABSTRACTIONS FOR SOFTWARE COMMUNICATION

PROGRAMMING PROTOCOLS IN TYPESCRIPT

by

Jan van Overveld

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University

Student number	851669922	
Faculty	Faculty of Management, Science and Technology	
Institute	Open University of the Netherlands	
Course	IM9906 Graduation assignment	
Master program	Master's Programme in Software Engineering	
Supervisor	dr. ir. Sung-Shik Jongmans	Open University
Chairman	Prof. dr. M. C. J. D. van Eekelen	Open University

ABSTRACT

More software parallelization results in more software communication and therefore the need for software communication protocols. Protocol logic in a program is also called the coordination concern. Separating this coordination concern from the regular computation concern in a distributed application can result in higher quality software code. This can be achieved by providing a software engineer facilities for programming protocols based on a global protocol definition. Not only will the coordination be separated from the computation concern, but the defined protocol can also be statically validated. Generating the coordination layer for a distributed application also saves time and reduces programming errors instead of implementing this layer manually.

This thesis reports on research for adding software protocol abstractions to the programming language TypeScript. TypeScript is a statically compiled and typed version of the programming language JavaScript, which is a popular language for programming distributed applications.

Various concrete abstractions that are needed for the protocol abstraction layer are divided over three conceptual layers. The first conceptual layer provides abstractions needed for starting the protocol and a role based messaging system. The second layer provides a programming interface for each distributed process that is part of the protocol. This interface will enable static protocol validation and with a professional development environment, facilities like code statement completion and code navigation will be present. The third layer provides the global protocol definition with the corresponding programming syntax, through an external DSL.

A runtime facility for starting up the protocol, to make sure all involved distributed software processes are present, is called the Mediator. This Mediator role collects the physical address details of all the protocol involved roles and delivers this information to these roles. After this startup process the Mediator dies and the protocol has started for the involved protocol roles. Messages are specified and can be continuously received, but processed when a role is ready for a specific message. Static protocol validation is based on state machine simulation per role. The generated API will have objects for all states, and the methods of these objects represent the transitions between states. These transitions can only be made once, which is checked at runtime to guarantee protocol safety. Performance penalty tests indicate a low decrease of performance due to the extra abstraction layer, and this happens only when the amount of messages increase to high volumes.

Normally the implementation of the coordination concern of a distributed application can be error prone, but with a generated coordination layer and static validation, protocol constraints are enforced and therefore errors are reduced.

CONTENTS

List of Figures

1

INTRODUCTION

It is very likely that software processes will become increasingly concurrent in the future, and that parallel processing of software will become more ubiquitous than it is already today. As the amount of parallel processing increases, the amount of communication between processes increases too, and when there is communication, there are communication protocols.

Parallel processing exists in the large, like distributed systems as for example the public Internet or a company enterprise system where several software services cooperate to deliver a piece of functionality. And parallel processing exists in the small, as for example the parallel execution of software on a single computer with a multi-core processor.

This first chapter of this Master thesis starts with an introduction in parallel processing in the large in the form of distributed systems in Section 1.1. In Section 1.2 a definition and explanation of a software protocol is given. Then in Section 1.3 the importance of separating concerns in software applications is mentioned concerning software protocols. Domain specific languages for programming software protocols are introduced in Section 1.4. Then the main research question of this Master thesis is formulated in Section 1.5 with the narrowed down sub questions. This chapter ends with an explanation of the contribution that is done with this research in Section 1.6.

1.1. DISTRIBUTED SYSTEMS

In computer science, a distributed system is a system that consists of different autonomous systems that cooperate and communicate by using message passing techniques. Distributed software systems are already omnipresent; nowadays, all software on Internet connected devices communicate with centralized or decentralized services. An upcoming architecture paradigm as Micro-services (MSA) also exemplifies trend towards distributed software, programmed as tiny little services [Dra+17a]. Notably, Dragoni [Dra+17b] argues that a software architecture based on micro-services is a natural fit for the Internet Of Things (IOT). According to Zhang [Zha16], at this moment, there are more than 20 billion connected devices, while trillions of Internet-connected sensor devices are expected to be connected within ten years. In addition, there are also indications, like the multikernel architecture [Pfe16], that computers with operating systems that treat their multi-core processors as distributed systems tend to be more scalable. This can also be thought of as using

message passing instead of shared memory and these ideas have already resulted in a few operating system implementations, like Barrelfish and Singularity [Pfe16].

An increase of parallelism among distributed software processes will likely result in an increase of communication between these processes. For successful communication between processes, software protocols are essential.

1.2. SOFTWARE PROTOCOL

A software protocol defines the admissible communication patterns between software processes; this includes infrastructure specifications to make communication possible and a description of the flow of the communication itself. The infrastructure specification includes physical network addresses with corresponding network protocols and a description of the exchanged message format. The flow description consists of the order of execution of the software processes and other behavioral constraints that define the behavior of an instance of the defined software protocol; this is also called a conversation.

As more software-services join the computing environment, distributed software execution will become increasingly important, and a lot of these services will participate in a software protocol to produce the desired functionality. The responsibility of the coordination of protocols can be handled within specific enterprise architectures, like SOA or MSA and their corresponding techniques, or it can be implemented directly in a program. If a piece of software is not a part of an enterprise, the latter will often be the most preferable option, because enterprise architectural facilities for coordination will be missing. Of course it is possible to implement distributed coordination logic directly into the software, but this increases complexity. Specifically, intermixing of different pieces of logic with different concerns, across the whole program code violates a software engineering best practice principal, called 'Separation of Concerns'.

1.3. SEPARATION OF CONCERNS (SOC)

SoC is a popular and well known principle for engineering software, the benefits have been known for decades within the field of computer science by Parnas [Par94] and also our own Dutch computer scientist Edsger Dijkstra [Dij79]. With this principle, programming logic becomes much more comprehensible and understandable for humans and therefore more maintainable.

When implementing software protocols, the coordination concern can be separated from the computation concern. This way programming code becomes more comprehensible and therefore easier to understand [GC92] and adjust.

Achieving better quality programming code which is more maintainable, is one of the main goals of this master thesis. Therefor this subsection deserves its own separate space. In this document there are references to this section to show the practical loyalty to this principal.

1.4. PROTOCOL LANGUAGES

As argued for many years [GC92; Cia96], in many papers and congresses [Cia+18], the separation of computation and coordination logic increases the quality and maintainability of software code in which a software protocol is implemented. A protocol language is a

programming language tailored to implementing the coordination concern of a software protocol. Examples of these kind of languages are Linda [CG89] and Reo [Arb04]. These two languages are capable of implementing the coordination part of parallel software processes on a single physical machine, with the use of shared memory (although distributed implementations exist as well). The revolution and evolution of multi-core processors from dual cores in 2004 to processors with eighteen cores at this moment, makes the development of these protocol languages an interesting research topic in software engineering. Writing parallel programs is not standard in most languages, and maybe a bit unnatural for many programmers who are used to sequential program processing. But the complexity of non-sequential programs is needed to fully use the multi-threaded capabilities of a modern processor.

There are also protocol programming languages in which the focus is on message passing for communication between software processes, instead of using shared memory. Examples of these kinds of languages are Jolie [MGZ14] and Scribble [Hon+11]. The idea is the same: a programming language specifically tailored to coordination of software processes is used to program the coordination concern. A programming language for a specific concern is also called a Domain Specific Language (DSL).

A software communication protocol between different distributed autonomous software processes, which are called the roles in the protocol, can be described on a global level. This way the complete constrained interaction between the roles is described in one program. Describing the protocol on a local level than the protocols are described per role separately. The protocol DSL Scribble is discussed in Chapter 2.1.3. This language describes from a high global perspective the interactions between the roles which can be seen as distributed processes. With Scribble a complete protocol can be defined, the syntax encapsulates the interaction of the different parties. This is a difference with the DSL Jolie discussed in Chapter 2.1.4 in which the protocol is defined by describing local perspectives. With Jolie the interactions of the different parties in the protocol are described separately and not globally as with Scribble.

One description of a multi party communication protocol on a global level is less work for a software engineer then describing each protocol for each role that is involved in the protocol. Also the higher abstraction level of a global description should make the interaction flows more comprehensive and therefore more understandable. A description of a software communication protocol on a global level is the preferred way of describing a constrained distributed software communication pattern for this Master graduation project.

A generated software communication framework based on a description of a global protocol, could be used by a software engineer to build his distributed application. Such an API generation process should be applicable and beneficial for any statically compiled language [Hu17]. The advantages of statically compiled languages in opposite of dynamic languages are for example the static validation of the protocol constraints and the ease of syntax completion facilities when using an appropriate IDE.

1.5. MASTER RESEARCH

As distributed processing becomes ubiquitous, facilities for describing protocols in programming languages become increasingly important. Existing GPL programming languages lack facilities for software protocols and are therefore inadequate.

Memory management in an software application once was the responsibility of the pro-

grammer. The introduction of the garbage collector changed this, and languages like Java took away this responsibility, making the programmer's job easier. Will this be a future scenario for software communication protocol abstractions, and is an evolution with these abstractions in programming languages a logical step forward? A field in Computer Science that researches the parallelization of software exists, and several domain specific programming languages have been developed for defining constrained software communication patterns.

A contribution to this research field is done with this Master project. This project aims to develop facilities for software protocols in the GPL language TypeScript. TypeScript is a typed version of JavaScript which is a frequently used distributed programming language in the world at the moment [hac18; bus18; fre18; use18; red18]. This way, with the help of TypeScript, static protocol syntax validation can be achieved for JavaScript. The protocol description is formulated with higher abstraction level definitions describing the allowed interactions between distributed processes.

The main research question proposed in this research proposal is:

How can protocol abstractions be added to TypeScript?

This research question can be narrowed down to five questions:

- RQ1: What abstractions do we need?
- RQ2: Do we need a protocol engine for validating and execution of protocol instances?
- RQ3: How is message passing implemented?
- RQ4: Is it possible to add these abstractions as GPL or will it better to add them as embedded DSL.
- RQ5: What is the performance penalty of these abstractions?

1.5.1. RQ1: WHAT ABSTRACTIONS

What are the concrete abstraction facilities that are needed for implementing the distributed software communication abstraction layer in TypeScript?

1.5.2. RQ2: PROTOCOL ENGINE

Is there a need for runtime facilities for execution and validation of software communication protocols in TypeScript? And if a runtime engine is required, for what is it needed and how is this runtime engine implemented?

1.5.3. RQ3: MESSAGE PASSING

How will the message passing between distributed software processes take place? What abstraction facilities are needed and how are these implemented?

1.5.4. RQ4: GPL OR DSL

What is the most convenient way for adding software communication protocol abstractions to TypeScript? Directly adding these abstractions to the language, and make them a part of the TypeScript GPL language, or implement these abstractions as a separate DSL. TypeScript is an open source language with its source code on GitHub, which makes it possible to directly modify the TypeScript language interpreter.

1.5.5. RQ5: PERFORMANCE PENALTY

What is the performance penalty of the execution of a distributed application that is built with the use of protocol abstractions? Generating the coordination concern of an application, this way separating the computation concern and provide help with building the distributed application could result in a loss of performance.

1.6. CONTRIBUTION

Generation of an API based on a global protocol description with the DSL Scribble to other static compiled languages has been done before. For example a process for generating a Java API is described in a paper by Hu [Hu17]. The uniqueness of this Master project is the generation of an API based on a software communication protocol description to the static compiled language TypeScript. There has not yet been a contribution in this field of software engineering that generates a protocol API in this language. Since the language TypeScript runs as JavaScript, which is the most used distributed programming language at this moment, this is a relevant contribution to the field of parallel distributed programming. The type system and static compilation offered by the TypeScript language in combination with the generated protocol API make static protocol validation possible for JavaScript.

In Chapter 2 there is background information about protocol abstractions and also the programming language TypeScript. Section 2.1 discusses protocol abstractions with concrete practical examples. Section 2.2 describes the programming language TypeScript and related technologies, since the goal is to add protocol abstractions to this GPL.

Chapter 3 explains the result of the this master project, where concrete abstractions are divided over three conceptual layers. With this solution software protocols can be programmed with static software protocol validation in TypeScript and executed as JavaScript. Section 3.1 starts with an example of a distributed program and show the bare implementation in TypeScript. Then Sections 3.2, 3.3 and 3.4 explain the conceptual layers. Chapter 3 ends with a conclusion in Section 3.5.

This thesis ends with Chapter 4 in which the result of this research is evaluated and the research questions are answered.

2

PRELIMINARIES

This chapter discusses background information concerning this project.

In Section 2.1 protocol abstractions are discussed. Concrete technologies for implementing distributed protocols were studied and documented in this section.

Protocol abstractions are added to the programming language TypeScript, therefor Section 2.2 introduces this general purpose language. Also other related technologies are discussed in this section.

2.1. PROTOCOL ABSTRACTIONS

Since the focus of this research is on message passing as communication technique for parallel software services, this chapter describes tools and techniques for the coordination of distributed software services. Techniques that can deal with the coordination of processes based on shared memory will not be discussed.

A big difference between architectures as SOA and MSA and implementing the protocol directly in the software, is the separation of the coordination responsibility and the computation responsibility. For better application management, separation is recommended because software code becomes more comprehensible, and because generic facilities delivered by external coordination solutions, like visualization or statistics, can help in the development of distributed processes. One can argue that another good reason for separating computation from coordination is the possibility to choose the most appropriate technology for the job that has to be done; for example, for a computation-intensive, job the C++ language can be used, while for coordination a language like BPEL could be chosen.

There are also several examples of DSL programming languages with which coordination of distributed software can be implemented by leveraging type systems. Scribble [Hon+11; Hu17] and Jolie [Ban+16; MGZ14] are examples of such DSL programming languages. In the book 'Behavioral Types in Programming Languages', behavioral types are discussed [Anc+16], and also the language Sing# is mentioned as a programming language in which behavioral types are embedded in the type system, but since this language is only for the operating system Singularity, it must be considered a DSL with a very specific scope. The Singularity operating system is a prototype of a reliable operating system where the kernel is treaded as a distributed system based on message passing communication, without shared memory.

In Section 2.1, first an introduction is given for behavioral abstractions and then three technologies for coordinating distributed services are described. The first one in Section 2.1.2 is the tool Oracle Workflow which is used in enterprise environments to coordinate business processes. This tool is a way to coordinate distributed services based on workflows and was used in some enterprise companies for distributed software processing before the SOA age. The second technology discussed in Section 2.1.3 is the coordination language Scribble; this is a DSL programming language for implementing software protocols. Section 2.1.4 deals with the language Jolie, also a DSL language specifically tailored to the implementation of coordination logic. Section 2.1.5 discusses the possibilities for architectures of software protocols.

2.1.1. BEHAVIORAL ABSTRACTIONS

Behavioral abstractions are facilities in technologies with which we can define the communication and behavior of concurrent software processes. These can be statements or datatypes in DSL languages for coordination or facilities in distributed architectures.

Software engineers all know primitive data types, like number, raw and character. But also complex object types, like objects or interfaces with which we can define higher level datatypes that are capable of representing the real world and the problem domain. Behavioral types can sometimes be encoded by creating custom complex datatype abstractions which are available in the used technology. Several programming languages also facilitate Aspect Oriented Programming (AOP) with for example the help of annotations or decorators. Functionality like for example logging is separated from the computation logic with the use of these AOP constructions.

To focus on what kind of abstractions are needed for protocol programming, three distributed service technologies are discussed in following sections.

2.1.2. ORACLE WORKFLOW

Oracle Workflow is a tool for defining enterprise business workflows; this was a familiar tool in Oracle-centric enterprises in the mid 2000s. Simply said, this was a gui tool that could visually represent and build workflows of processes, and with this tool, a text file was generated wherein a specific execution-order of and communication between distributed software was described. An engine took subsequently care of the execution of the software and the validation of the behavioral constraints.

The technology behind this tool has become deprecated and unsupported. The more generic SOA architectural paradigm took over, with its supporting tools and technologies. In a SOA architecture, typically, the coordination of distributed software is implemented with the use of the Business Process Execution Language (BPEL). And nowadays, we have entered the age of the MSA architectural paradigm [Dra+17a] with its facilities for distributed service coordination. An interesting example is the custom MSA architecture from Netflix, called Conductor [Net18].

The Oracle Workflow technology was capable of coordinating distributed processes, implemented using many forms of different technologies. The generated workflow could contain processes, other workflows and logical operations, like routers or loops. Thus, a process could be a workflow, a programming method, a unix shell script, interaction with a user with the use of a graphical screen, and interaction with other systems based on for example remote procedure calls or protocols like the File Transfer Protocol (FTP). The in-

teraction of the processes with each other in the same workflow was dealt with by the exchange of predefined variables; these communication variables were specified in the workflow template. Of course, the workflow engine had some variables of its own too, which made it possible to identify specific instances of workflows and collect some management information of instances of workflows like timing.

When looking at this technology as an example of coordinating distributed processes in enterprise environments in the old days, it is clear that several components can be extracted. These components can be classified in two groups, namely metadata and physical data. There is metadata about workflows, processes in a workflow, and the interaction between the processes in the workflow, which could be called a message interface (i.e. an agreed upon contract between processes). The information about processes include information as the kind of process, if relevant the used protocol, and the interface it offers. A workflow is nothing more than the predefined coordination of these processes, while the instances of a workflow are the physical executions of this workflow.

2.1.3. SCRIBBLE

Scribble is a DSL programming language for implementing the coordination concern; its purpose is to provide a programming language for implementing protocols between distributed processes [Hon+11; Yos+13; Hu17]. Scribble implements protocols and can also describe an architecture of protocols. The computation concern is dealt with in the called services with their own implementations and programming languages. These services are called participants while the instantiation of a protocol, including the exchange of messages between two or more participants, is called a (multiparty) conversation or session. The description of the flow of these messages is called the protocol.

The formal and intuitive Scribble language makes it possible to reason about communication protocols. In Figure 2.1 [Hon+11] an overview of the Scribble software framework is visualized.

Distributed processes can implement interaction through the conversation API. These are interfaces for passing messages between processes in the described protocol; in Scribble, these are called the messages in the conversations. Static validation is carried out with the aid of the API. The runtime checking is done by the Runtime Monitor; this monitor reads the protocol specification and inspects the runtime conversation behavior.

Let's look at an example protocol named Alice and Bob (1), a client-server protocol for a network service that adds two numbers. This example protocol will be described in the Scribble language syntax in the rest of this section. This example is based on the Adder example that is used on the Scribble website [Scr18].

(1) The outline of the protocol is: Alice may choose to send to Bob one of the following two messages, an ADD message or a BYE message. The ADD message has a payload of two Integers and after receiving the ADD, Bob sends to Alice a RES message with a payload of one Integer (this is the sum of the received Integers). Then Alice and Bob continue by looping back to the start of the protocol. A BYE message with an empty payload will end the conversation. The protocol ends for Alice after sending the BYE message, and the protocol ends for Bob after receiving the BYE message.

A sequence diagram illustrating the global perspective of the Alice and Bob protocol example (1) is displayed in Figure 2.2.

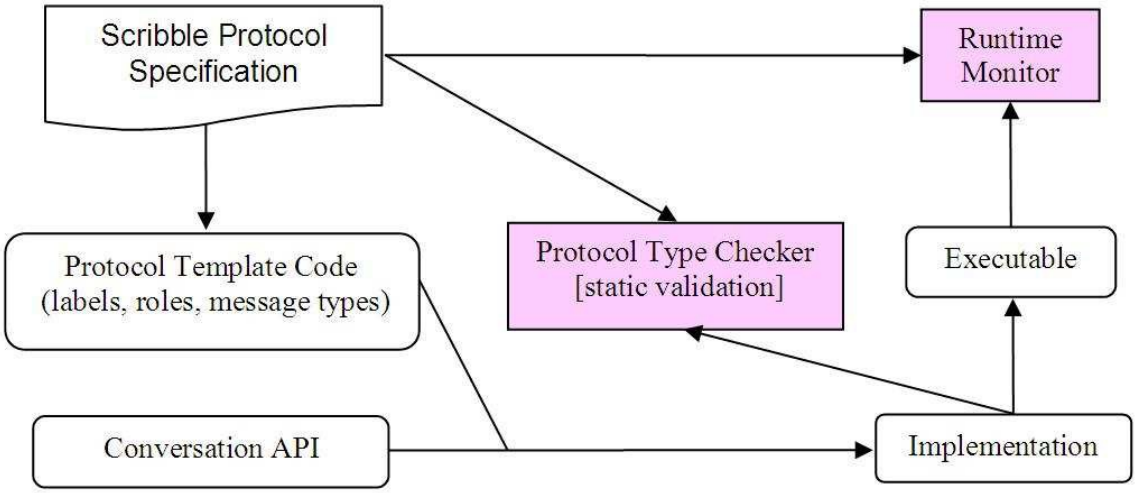

Figure 2.1: Scribble framework overview

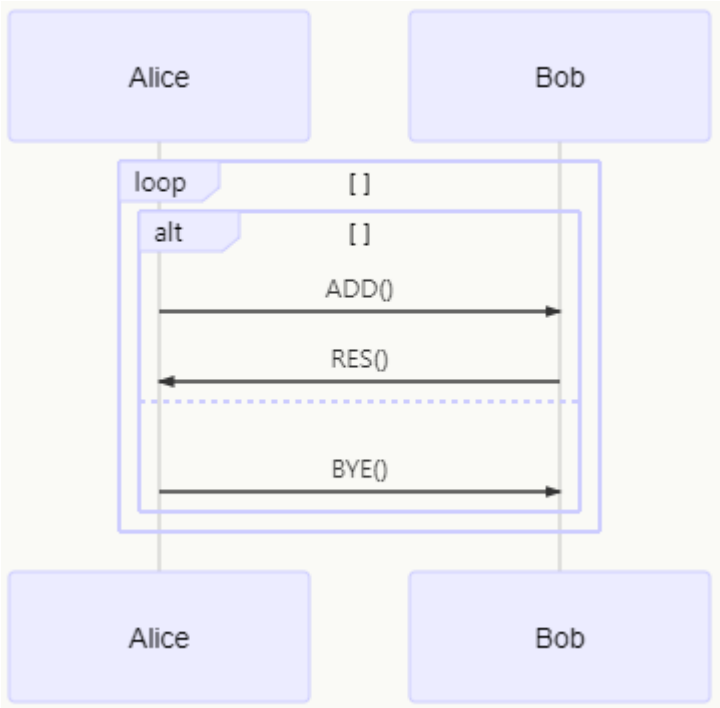

Figure 2.2: Alice and Bob global protocol

Listing 2.1: Scribble code for implementing Adder software protocol

```
1
module examples.AliceAndBob;
3
type <java> "java.lang.Integer" from "rt.jar" as Int;
5
global protocol AliceAndBob(role Alice, role Bob) {
7     choice at Alice {
          ADD(Int, Int) from Alice to Bob;
9         RES(Int) from Bob to Alice;
          do AliceAndBob(Alice, Bob);
11    } or {
          BYE() from Alice to Bob;
13    }
}
```

The code example in Listing 2.1 is an implementation of the Alice and Bob software protocol with the Scribble DSL; this implementation follows the description of example (1). The code starts with a module declaration on the first line; after that the type for the exchanged message is defined. Then, the protocol specification begins with the protocol keyword. AliceAndBob is the name of the protocol and the curly braces indicate the protocol body; the two participants Alice and Bob are defined with the role keyword and are specified as parameters of the protocol. The choice statement states that the protocol has a choice between the two separated blocks of code In this example, Alice is the master role that decides which block to follow. Alice communicates its decision to the participant Bob by passing corresponding messages in each case. The message signature ADD(Int, Int) specifies that ADD is the operator (this is a label or header that identifies the message) and the payload is two integers (Int,Int). In the message passing statement 'ADD(Int, Int) from Alice to Bob;' Alice is the sender, which without blocking, asynchronously dispatches the specified message; in turn, Alice is the receiver and blocks until the message is received. The do statement 'do AliceAndBob(Alice, Bob);' allows a protocol to be defined in terms of other (sub)protocols. This makes the creation of an architecture with multiple protocols possible. But it is also used for recursive protocol definitions, and can be used for the iteration in the AliceAndBob protocol example.

This Scribble protocol definition can be parsed with the Scribble compiler, and after a successful validation, the monitor code is generated. The generated endpoints are based on Finite State Machines (FSM) [Hu17; Scr18], see Figures 2.3 and 2.4. Physical addresses like hostnames and ports of distributed services have to be passed through with the initialization of a protocol. These can be passed with the constructors of the generated code for the participants.

In the FSM examples in Figures 2.3 and 2.4, the transitions are the exchange of the messages. A state sends or receives a message to proceed to a next state.

2.1.4. JOLIE

Jolie is a programming language for coordinating distributed processes. A lot of the literature on Jolie is also about the MSA architectural paradigm and how this language is the best

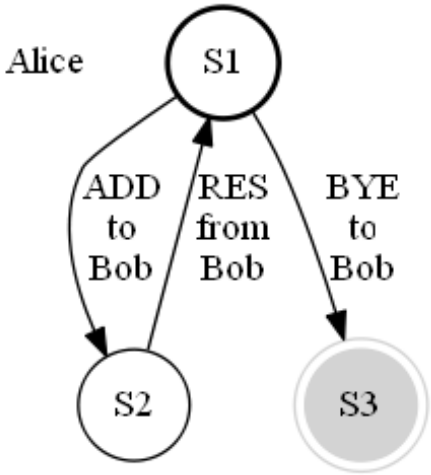

Figure 2.3: Local perspective of Alice

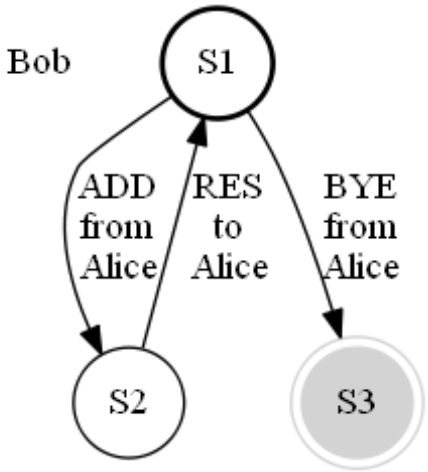

Figure 2.4: Local perspective of Bob

natural fit for this relatively new architecture [Dra+17b; Gui+17; MGZ14; Ban+16; GDM16; MGZ14; Tch+16].

Jolie is an abbreviation of Java Orchestration Language Interpreter Engine and is an open-source programming language. The Jolie project started in 2006 by Fabrizio Montesi as part of his studies at the University of Bologna. In Jolie, separation of behavior and deployment is a major design feature. Deployment information includes physical addresses at which functionalities are exposed and the communication technologies and data protocols used to interact with other services.

The example protocol Adder (1) from the previous section will be used to illustrate the implementation code with the corresponding abstractions in the Jolie programming language.

In the Listing 2.2, an interface for the distributed service is defined; this is the first abstraction in the Jolie language. The interface defines the messages that will be exchanged during the conversation; this code listing follows the Adder example (1).

Listing 2.2: Implementation of a message interface in the Jolie language

```
interface AdderInterface {
    RequestResponse: add( int, int )(int)
    OneWay: bye
}
```

As said before, in Jolie, deployment is separated from behavior. For deployment abstractions, there are inputPorts and outputPorts in which physical details of services can be defined. These physical details include port number, domain names, IP addresses and low level communication protocols. Following the Adder example of (1) the syntax for the inputPort is shown in Listing 2.3. This code represents the server part of the protocol.

Listing 2.3: Implementation of an inputPort in Jolie

```
include "AdderInterface.iol"

```

```
inputPort AdderService{
    Location: "socket://www.AdderSolutions.nl:8000/v2/add"
    Protocol: https
    Interfaces: AdderInterface
}
```

The outputPort is also a deployment abstraction, Listing 2.4 contains the physical information for the client part of the protocol.

Listing 2.4: Implementation of an outputPort in Jolie

```
    include "AdderInterface.iol"

    ouputPort AdderService{
        Location: "socket://localhost:8000"
        Protocol: https
        Interfaces: AdderInterface
    }
```

The inputPort and the outputPort are connected to the predefined AdderInterface interface. This way the message that is exchanged is connected to the physical service. The complete code for the server logic is shown in Listing 2.5.

Listing 2.5: server side implementation in Jolie

```
    include "AdderInterface.iol"
    inputPort AdderService{
        Location: "socket://www.AdderSolutions.nl:8000/v2/add"
        Protocol: https
        Interfaces: AdderInterface
    }
    main
    {
        [ add( inputX, inputY )( result ) {
            result = inputX + inputY;
        } ]
        [ bye {
         // clear session  etc...
        } ]
    }
```

The Jolie implementation code for the client side of the software Adder protocol is shown in Listing 2.6.

The recursive iteration in the Scribble code in Listing 2.1 with the use of a sub-protocol on line 10 can not be achieved the same way in Jolie. In Jolie, this iteration is achieved with the statements provide until: the add service will be called until the bye service is called.

Listing 2.6: client side implementation in Jolie

```
include "console.iol"
include "AdderInterface.iol"
ouputPort AdderService{
    Location: "socket://localhost:8000"
    Protocol: https
    Interfaces: AdderInterface
}
main
{
    provide
    [ add( inputX, inputY )( response ) {
            add@AdderService( inputX, inputY )( response );
        }
    ] { println@Console( response )() };
    until
    [ bye {
            bye@AdderService( );
        }
    ] { println@Console( "Session␣ended" )() };
}
```

Interestingly, whereas in Jolie deployment is separated from behavior, in Scribble these physical details are parameterized and passed during object initialization. One can say a developer has more flexibility this way, and can make its own deployment details solution, without the constraint to wrap it in an inputPort object. However, Jolie forces developers to think about deployment, and this is essential when services are distributed. Oracle Workflow just as Scribble, did not have separate objects for the deployment characteristics of distributed services, these were also properties of process objects.

2.1.5. ARCHITECTURE OF SOFTWARE PROTOCOLS

An architecture of protocols is a set of multiple protocols combined; this is enabled by architectural styles like MSA and SOA, but there are also protocol languages that are capable of defining software protocol architectures. Examples of these programming languages are the language Chor [CM13] and the framework Adaptive Interaction Oriented Choreographies in Jolie (AIOCJ) [Dal+14]. AIOCJ and Chor both generate Jolie programming code, but AIOCJ can be adjusted runtime, which is often essential in enterprise production environments. This in theory makes AIOCJ a suitable candidate for MSA choreographies in production environments. Scribble is also capable of defining an architecture of protocols with the use of subprotocols.

In this research proposal, the focus will be on distributed protocols where the communication is based on message passing; architectures of protocols will be out of scope.

2.2. TYPESCRIPT

Atwood's Law: any application that can be written in JavaScript, will eventually be written in JavaScript [cod18].

In this chapter, the GPL programming language TypeScript (TS) is described. The language TS runs as JavaScript (JS). JS is one of the most popular programming language at this moment [hac18; bus18; fre18; use18; red18]. JS language interpreters run on almost every computer, game console, tablet, and smart phone, and is used almost always in a distributed setting. TS extends this language with typed data and more, the language is actively developed, and many programming facilities that are known from other GPL programming languages have been integrated in the TS language. TS is also capable of producing JS that is compatible with most browsers. The popularity and distributed nature of JS, and the sophisticated extras of TS make this GPL a highly suitable candidate to be extended with facilities for distributed protocols.

TypeScript is statically compiled and has an advanced type system. This makes static compilation of distributed protocol constraints possible. Also with the use of an advanced IDE statement completion for protocol programming will be possible.

Section 2.2.1 gives an introduction on TypeScript. The JS programming language will be discussed in Section 2.2.2. More detailed information about TS follows in Section 2.2.3 with some coding examples. Section 2.2.4 discusses Node.js, this is the server side solution for JS. Section **??** ends with some remarks which will end this chapter.

2.2.1. INTRODUCTION

TS is an open-source GPL programming language that is originally developed by Microsoft [Mic12]. After a development cycle of two years, the language was released in October 2012 as open source software, which made it possible for everybody to help with further developing the language. The first version of TS was 0.8 and at the moment of writing version 3.1 was already published.

TS is a superset of JS, all JS code compiles as TS, but TS has many language features that are not part of JS and therefore TS cannot be interpreted as pure JS.

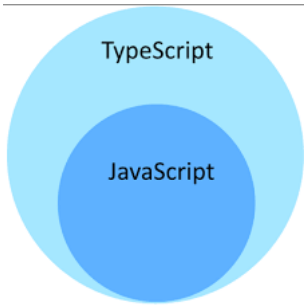

Figure 2.5: TypeScript a superset of JavaScript

Since the target runtime environments for TS code are JS interpreters, TS code is transpiled into equivalent JS code. Transpilation is the process of converting code written in one programming language into code written in another programming language at the same level of abstraction. When a programming language is translated into a lower level programming language, this process is called compilation; this is the case when compiling code to assembly code.

2.2.2. JAVASCRIPT

For TS to be run by a JS interpreter it is first transpiled to JS. JS is a well known programming language that was originally developed around 1995 for the Netscape browser, one of the first Internet browsers. The language was conceived to give websites more dynamic content, also known as Dynamic HTML (DHTML); it was a scripting language for manipulating the Document Object Model (DOM) of a browser. But the language has evolved over the past two decades into a widely used language [Haf+16]. Widespread support for the language by all Internet browsers has made JS the standard language to develop advanced websites that run on all computers without extra software required, contrasting for example a Java Runtime Environment.

Standardization of the specifications of the JS language with the help of the European Computer Manufacturers Association (ECMA) succeeded after a period of ten years The ECMA specification for JS is called ECMAScript; JS is an implementation of a ECMAScript specification. At the moment of writing, the JS language is the most popular language on GitHub [hac18; bus18; fre18; use18; red18].

JS code is interpreted instead of compiled; this means the translation of the programming code to a lower level representation is performed at runtime. Syntax and data type checking are executed at runtime; in contrast, languages like Java and C# do this before execution. This makes JS a scripting language with a lot of data type flexibility; for example, variables can even change data type during execution. Or arrays can consist of different elements with different data types, like number, boolean, object and even functions. A dynamically interpreted language with no type checking is quite flexible in a way, but it has some downsides too. For example having no strict data types in a programming language makes it hard to develop professional Integrated Development Environments (IDE) which makes the life of software engineers harder. An IDE with code completion is quite useful in a language like JS for which a huge amount of different libraries are written. Especially software engineers that lack the necessary experience in a specific library will have improved productivity when the objects and functions are known and described in the IDE. Also static code analysis for security, code quality or performance is harder to implement with an interpreted language that has no static typing of its data.

But besides the lack of type checking in JS, the programming language has some other problems too, like difficulties with variable scoping, confusion about iterations on collections, deprecated problematic syntax like the with statement, complex object orientation syntax, and unexpected behavior of the keyword **this**. The 'use strict' declaration was introduced in ECMAScript version 5 to prevent the use of deprecated features and prevent other actions like the use of undeclared variables [Haf+16]. Generally with every release of a new ECMA version, problematic issues are taken care of and new features to improve the language are introduced.

Functional programming and object orientation are both possible in the JS language. Moreover, functions are first class citizens: they can be given as a parameter of a function or be the return type. Object orientation is implemented as prototype inheritance instead of class-based inheritance, which is used in languages like Java or C#. Simply said, prototypical inheritance means that every object has a prototype property that points to its parent object. From ECMAScript 6, it is possible to use class-based syntax for object orientation in JS, but underneath, it is translated to prototypical inheritance. However, the syntax is much more familiar for most programmers [Sil+17].

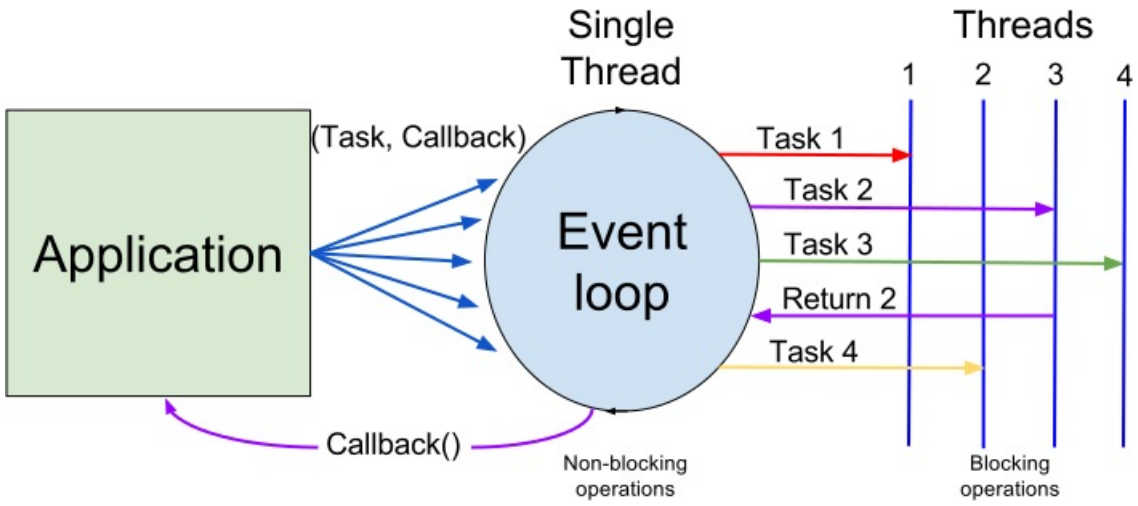

Figure 2.6: The JavaScript eventloop
[out18]

JS is also known for its non-blocking architecture, which is called the Event Loop. In Figure 2.6 a visual representation of the Event Loop is illustrated [out18]. This Event Loop architecture for JS is active in a JS engine in a web-browser, but also on a Node.js server executing serverside JS code; a section about Node.js will follow.

In the Event Loop architecture, there is **one** main thread that runs inside the JS runtime environment. This thread takes events from the DOM in the case of an interpreter that runs for example in a Chrome web-browser. An event from the DOM is a piece of JS code that is connected to HTML elements and their corresponding JS actions, like click, mousemove, mouseenter etc. In the case of a server application running on Node.js, the JS engine runs a JS application, as shown in Figure 2.6.

The important point here is that there is one main thread for handling the events or executing the JS application code. This architecture ensures that developers are forced to use the asynchronous facilities that are present in JS. These asynchronous facilities are executed in background threads separate from the main Event Loop thread. This Event Loop architecture logically results in asynchronous non-blocking code. In other programming languages like Java, asynchronous execution is not normal as in JS, and more effort has to be taken to achieve this. The main thread executes all events and callbacks that are in the queue sequentially following first in first out pattern.

In JS, programmers can use asynchronous facilities to submit code to background threads. In the client, this is done with asynchronous APIs like XML HTTP Requests (XHR), also called AJAX (Asynchronous JavaScript And XML), and other APIs and libraries like Web Workers, which is a way to manage background processes in the browser. In Node.js, serverside JS, almost everything is by default asynchronous, like writing and, reading to files and databases. The way to deal with asynchronous program code in the Event Loop architecture is with the callback programming pattern. When an asynchronous task is submitted to be executed, a piece of code is given as a parameter to be executed by the Event Loop after completion. This pattern uses the power of passing functions to other functions as parameters.

A relatively simple example of the execution of this Event Loop with the callback pat-

tern is given in code Listing 2.7; this example illustrates the power of callbacks with the use of asynchronous code. The program in Listing 2.7 is a sequential example: first console.log('start') is executed, then the function go with the function callbackFunction as callback parameter, and then console.log('eind'). The output in Listing 2.8 shows that first console.log('start') and console.log('eind') are executed, and last function go with the callback parameter, function callbackFunction. This is because the callback function is placed on the work queue of the Event Loop, with the help of setTimeout for asynchronously executing a function. First the statements of the program are executed, and then the work of the queue is handled by the main Event Loop thread. This is the code pattern to deal with asynchronous code in JS and therefore TS.

Listing 2.7: callback code pattern

```
function go(callback) {
    callback.call(this, 'Voorbeeld Argument');
}
function callbackFunction(arg) {
    setTimeout(function () { return console.log(arg); }, 1);
}
console.log('start');
go(callbackFunction);
console.log('eind');
```

Listing 2.8: output of callback code pattern example

```
start
eind
Voorbeeld Argument
```

When blocking operations are executed asynchronously, and callback code is coupled to deal with the actions followed after the completion of the asynchronous code, the Event Loop does not have to wait. Often in JS, blocking operations have asynchronous APIs which makes them non-blocking for the main thread this way. When an asynchronous blocking operation succeeds, the callback is pushed on the queue of the Event Loop to deal with the results of the successful asynchronous operation. Of course, there are also callbacks possible for unsuccessful asynchronous execution; this follows a similar callback pattern.

So it is relatively easy to implement asynchronous functionality in JS and pass a callback as a parameter. With the Event Loop architecture, and the corresponding programming style, the idle time of the cpu is minimized. Of course it is easy to block the main Event Loop thread with large computation, but this blocking piece of code can usually be refactored to an asynchronous function with a callback parameter, so the main Event Loop thread can continue working with non-blocking code. There are also downsides to this style of programming, like the callback hell that arises. A callback hell is a chain of nested callbacks which is very difficult to understand and maintain. Newer ECMAScript versions have a promise API to deal with this callback hell.

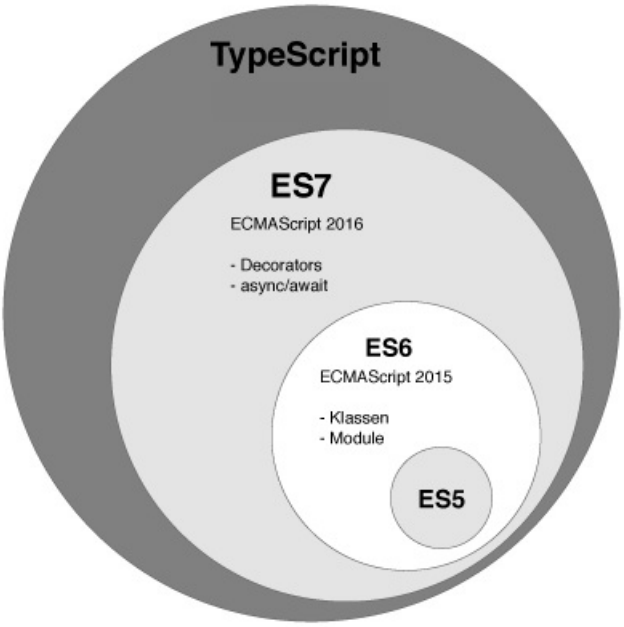

Figure 2.7: TypeScript a superset of ECMAScript

2.2.3. TYPESCRIPT, JAVASCRIPT THAT SCALES

On the website of TS [Mic12], TS is described as, 'TypeScript, JavaScript that scales'. Indeed, the primary reason why TS was created, was to make professional software development for big JS applications possible, manageable and maintainable.

A big issue is that to write JS code that most browsers support, JS code has to be written for interpreters in browsers for older ECMA standards; most Internet devices will not be equipped with the most recent JS interpreters. With TypeScript, it is possible to write code with the newest ECMAScript syntax and TypeScript features and transpile the code to the desired ECMAScript version. Beside static type checking, this is also a feature that increases the productivity of a software engineer who wants to work with all the recent features. As said before, TS is a superset of JS, but to be more specific, TS is a superset of ES7, ES7 is a superset of ES6, and ES6 is a superset of ES5; see Figure 2.7.

In code Listing 2.9, a simple program coded in TypeScript is given; this program has an interface, a class definition, an arrow function and the const keyword. These are all language features that are not yet available in ECMAScript version 5.

Listing 2.9: a TypeScript code example

```
interface ArgumentInterface {
    giveArgument():string;
}

class ArgumentSupplier implements ArgumentInterface {
    private argument:string;

    constructor(arg:string){
```

```
9           this.argument = arg;
        }
11
        giveArgument():string {
13          return this.argument;
        }
15  }

17  const arg:ArgumentInterface = new ArgumentSupplier('A nice Argument');

19  setTimeout( ()=>console.log(arg.giveArgument()), 1);
```

After transpilation of the TypeScript code to JavaScript code following ECMAScript standard 5 (this is a version that is supported by most JS interpreters) the code in Listing 2.10 is produced.

Listing 2.10: The corresponing transpiled JavaScript following ECMAScript 5 specifications

```
1   var ArgumentSupplier = /** @class */ (function () {
        function ArgumentSupplier(arg) {
3           this.argument = arg;
        }
5       ArgumentSupplier.prototype.giveArgument = function () {
            return this.argument;
7       };
        return ArgumentSupplier;
9   }());
    var arg = new ArgumentSupplier('A nice Argument');
11  setTimeout(function () { return console.log(arg.giveArgument()); }, 1);
```

2.2.4. NODE.JS, ONE LANGUAGE TO RULE THEM ALL

As said before, JS was originally developed to run on the client, in the web browser, and used for DOM manipulation. Around 2010, JS became available for application servers as well, in the form of Node.js; this made it possible to implement server-side logic in the same language as the client. The idea to run JS on the server was not new, and there were other implementations as well, but Node.js has become the most popular one. The package manager of Node.js, which is called Node Package Manager (NPM) is even more popular than the well known Maven package manager for Java at this time of writing. It has more than 800000 software packages available on the NPM website [npm18].

Node.Js made it possible to implement certain functionality in JS which was not possible before, like reading and writing to the file-system, or querying databases. Besides the same language, also the non-blocking architecture of the JS interpreter was copied to the server; this non-blocking architecture is called the Event Loop and was discussed in the previous section.

TypeScript is a Node.js package, which means that for using TypeScript, Node.js has to be installed too. There are some IDE tools that support TypeScript out of the box, but the

regular way of installing the TypeScript transpiler is to install the NPM package. An IDE like Visual Studio Code detects the availability of TypeScript as a NPM module and uses this software.

Since the popularity of JS, its practical field in the distributed software world and the ideas and professional state of the art technology behind TypeScript, this language was chosen as GPL programming language for this project.

3

RESULTS

Programming the coordination concern between software processes can be a complex job. It involves communication between partners that do not share resources, these partners are also called the roles in the software communication process.

The programming of the communication interactions between distributed processes is often done on relatively low level. By for example on the lowest protocol level with the use of sockets only using tcp/ip, and directly connecting and receiving data between machines. Or for example on a little higher level with the use of the http protocol on top of tcp/ip, constructions for remote procedure calls can be made. This is already a higher and more easier level than the level of sockets. Because instead of directly pushing bytes to another socket, and having the responsibility of making the right byte conversions yourself, this is taken care of in the http protocol. But both these ways of communication involve sending and listening for communication messages, that have to be composed and extracted. This is a lot of work and can be error prone, a violation of a rule that says one message must be received before the other is easily made.

To ease this process of programming parallel distributed communication between different autonomous software processes and facilitate in interaction constraints three architectural abstraction layers are proposed.

The first architectural abstraction layer is placed upon the low-level tcp/ip communication facilities that are already present. This layer will supply facilities to make role based communication possible. Role based communication means facilities for easily sending predefined messages to the predefined parties of the software communication protocol. The roles and the messages of the protocol are defined in the protocol description and therefore known, this layer delivers the tools for directly communicating with the roles. Constructing the low-level messages and physical addresses of the roles every time, is no longer needed. This first layer will be based on the roles and the messages from the protocol. Messages can be sent to roles involved in the protocol and messages will be received by roles from other roles and stored. This way these messages can be fetched by processes that expect and want a specific message on a specific moment. This layer uses the base tcp/ip facilities for communicating with other software processes.

The second abstraction layer will come on top of the first abstraction layer and will facilitate in static validation of the defined protocol constraints. This layer will deliver an API for the software engineer with which the distributed application can be built. Protocol

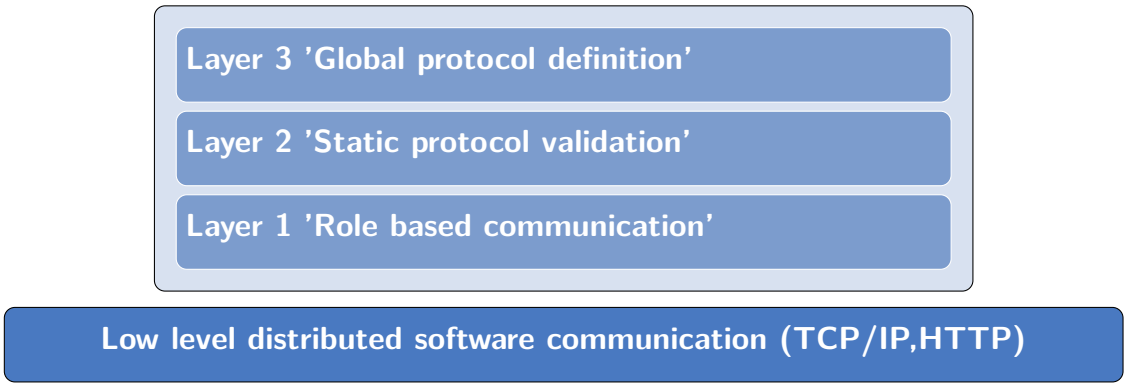

Figure 3.1: Layered abstraction architecture for distributed software communication

constraints like the the order of messages will be statically checked. With the use of a more advanced IDE, facilities like auto statement completion, code navigation or code refactoring are also possible. This second layer is based on a textual JSON definition describing the protocol. This description contains a textual representation of the FSM of all the different role perspectives that are involved in the complete protocol.

The third abstraction layer comes on top of the second layer, and supplies the second layer with the different local perspectives of the involved roles of the global protocol. In this third layer a global description of the flow of the messages between the roles in the protocol is formulated. This global description will be used to generate the JSON file with textual representations of the FSM of the involved roles. This file is used to generate the second layer with the static protocol validation facilities.

In Figure 3.1 these architectural abstraction layers for easing the process of implementing distributed software application are visualized. These layers are built on top of each other, every layer needs the layer beneath it.

In this chapter these three conceptual layers are discussed. A simple example that will be used in following sections is explained in Section 3.1. This example is provided with the TypeScript implementations without using generated protocol facilities. Referring to Picture 3.1, this section discusses an implementation with only facilities from the base layer 'Low level software communcation'. Section 3.2 will then discuss the first abstraction layer, and show an implementation based on the facilities present in this layer. Section 3.3 deals with the second abstraction layer and the used example is implemented with the facilities from this layer. Section 3.4 explains the third abstraction layer and shows the global protocol definition from the example. Finally there will be some additional comments in Section 3.5 and this will end this chapter.

3.1. ALICE AND BOB

A protocol code generation framework that generates an API based on a global description of a distributed protocol. This API will validate the distributed communication constraints from the specified global protocol. This will not only provide static protocol validation facilities but also give the programmer a head start developing the distributed application. Also separation of the coordination concern from the computation concern is enforced, which will make the application more maintainable. To illustrate this conceptual description of a protocol generator the example 'Alice and Bob' (1) from Chapter 2.1.3 is used. In

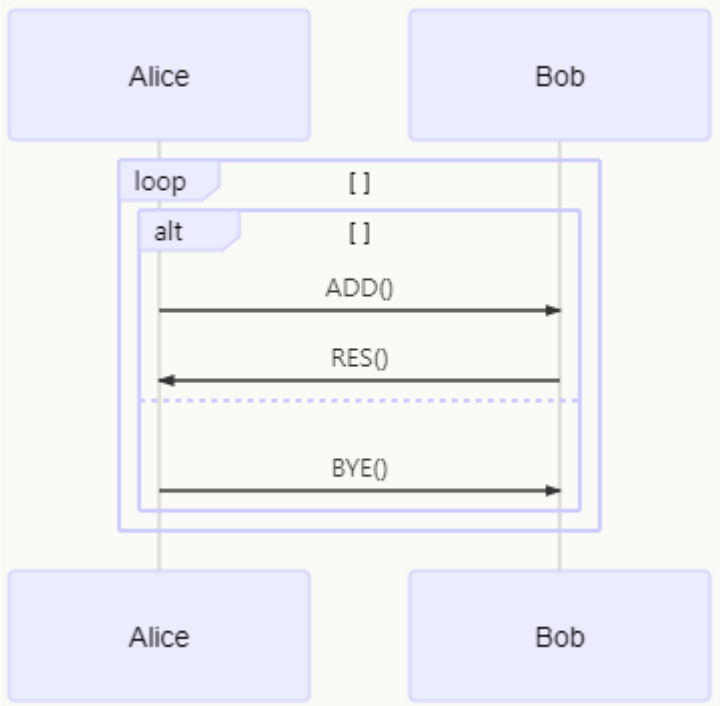

Figure 3.2: Alice and Bob global protocol

Chapter 2.1.3 Figure 2.2 shows the UML sequence diagram of the global protocol of the 'Alice and Bob' example, a global description of this protocol with higher level abstraction syntax is shown in Chapter 3.4 Listing 3.14. The local perspectives from different roles in the 'Alice and Bob' example can be illustrated with STD diagrams, one for the role Alice and one for Bob. These local perspectives are illustrated in Chapter 2.1 Figure 2.3 and 2.4. To reduce the page turns the Sequence diagram of this example is shown again in Figure 3.2. A visualization of the flow of messages between the Alice and Bob roles simplifies this example.

In Listings 3.1 and 3.2 a complete implementation of the Alice and Bob example in TypeScript is shown. This implementation is a standard implementation without the use of a protocol API and a possible way to follow when no protocol generation facilities are available. Each message is an object with a name property that serves for the identification. So when a message object needs to be identified the name property can be used. The object with the name ADD has two additional number properties. The object with the name RES has one number property and the BYE object has no properties. These anonymous objects with name identifiers are the messages that are sent during the protocol conversation.

There are two distributed processes, one called Alice and one called Bob. In Listings 3.1 the implementation of the autonomous Alice process is given. The implementations of Alice and Bob start with importing two standard JavaScript libraries. The first library is the http library and is used for creating a http listener. The second library is the request library and is used to send a http message to a http server. These JavaScript libraries facilitate in the sending and receiving of message objects between the two processes.

Listing 3.1: TypeScript implementation of Alice

```
import * as http from 'http';
import * as request from 'request';

const httpHeaders = {'cache-control':'no-cache'
                     ,'Content-Type':'application/json'
                          ,'charset':'utf-8'};
var   messageResolver: (msg: any) => void;

const httpServer:http.Server = http.createServer(
  (req,res) => {
     let body = '';
     req.on('data', (chunk:string) => body += chunk );
     req.on('end',  () => { messageResolver(JSON.parse(body));
                            res.write("OK");
                            res.end(); } );
  }
);

httpServer.listen(30001);

async function waitForMessage():Promise<any>{
   let promise = new Promise<any>( resolve => messageResolver = resolve );
   return promise;
}

async function sendMessage (host:string, port:number,msg:any) {
    let resolver: () => void;
    const promise = new Promise( resolve => resolver = resolve );
    const httpInfo = { url: `http://${host}:${port}`
                  , headers: httpHeaders
                     , body: msg
                     , json: true };
    request.post( httpInfo, () => resolver() );
    return promise;
}

async function startProtocol() {
   const hostBob='localhost';
   const portBob=30002;
   for(let i=0;i<5;i++) {
      sendMessage(hostBob, portBob, {name:"ADD", value1:21, value2:21});
      const res = await waitForMessage();
      if (res && res.name === "RES" && res.sum)
         console.log(`Received ${res.sum}`);
   }
   await sendMessage( hostBob, portBob, { name:"BYE" } );
   console.log('the protocol stops for Alice');
   httpServer.close();
}
```

```
51  startProtocol();
```

In the code listing from Alice 3.1 and Bob 3.2 from the first line to the function startProtocol all the code is responsible for the coordination concern of the program. A http server is created on line nine, and the listening starts on line nineteen. A function for receiving messages is created on line twenty one, which is called waitForMessage. This funcion returns a promise for a message, that is received by the http server. A promise is an object that corresponds with an asynchronous function that has not yet been fully executed. When an asynchronous function resolves the promise, the object inside the promise can be retrieved. On line twenty six the function sendMessage is created, which is used to send an http message to another distributed software process.

The async function startProtocol implements the computation logic from the protocol for Alice. Some low-level communication details are required here like the hostname and portnumber of Bob. Also validation checks on the message on line 43 have to be done to let the TypeScript code compile. In the startProtocol function a loop of five iterations exists on row 40, in this loop an ADD object message is sent to Bob. Then Alice waits for a RES message from Bob, the message is checked for a name string with the value RES and a valid sum property. After receiving this RES message, the message is displayed in line 44. After five iterations Alice sends an object with the string BYE to Bob to indicate the protocol ends and then stops the http server on line 48.

In Listing 3.2 the TypeScript implementation of Bob is shown. Bob starts listening on port 30002 on line nineteen. After executing the startProtocol function on line forty nine, Bob starts waiting for messages in the function startProtocol. When an ADD is received, Bob sums the numbers and sends a RES back with the result. When a different message than ADD is received, the protocol stops for Bob.

Listing 3.2: TypeScript implementation of Bob

```
1  import * as http from 'http';
   import * as request from 'request';
3
   const httpHeaders = {'cache-control':'no-cache'
5                       ,'Content-Type':'application/json'
                        ,'charset':'utf-8'};
7  var   messageResolver: (msg: any) => void;

9  const httpServer:http.Server = http.createServer(
     (req,res) => {
11     let body = '';
       req.on('data', (chunk:string) => body += chunk );
13     req.on('end',  () => { messageResolver(JSON.parse(body));
                              res.write("OK");
15                            res.end(); } );
     }
17 );

19 httpServer.listen(30002);
```

```
21  async function waitForMessage():Promise<any>{
       let promise = new Promise<any>( resolve => messageResolver = resolve );
23     return promise;
    }
25
    async function sendMessage (host:string, port:number,msg:any) {
27      let resolver: () => void;
        const promise = new Promise( resolve => resolver = resolve );
29      const httpInfo = { url: `http://${host}:${port}`
                    , headers: httpHeaders
31                     , body: msg
                       , json: true };
33      request.post( httpInfo, () => resolver() );
        return promise;
35  }

37  async function startProtocol() {
       let msg = await waitForMessage();
39     while ( msg && msg.name !== "BYE" && msg.value1 && msg.value2 ) {
          console.log(`Received ${msg.value1} and ${msg.value2}`);
41        const res = Number(msg.value1) + Number(msg.value2);
          await sendMessage('localhost', 30001, {name:"RES", sum:res} );
43        msg = await waitForMessage();
       }
45     console.log('the protocol stops for Bob');
       httpServer.close();
47  }

49  startProtocol();
```

The developer of the distributed application will have to know the protocol thoroughly. Mistakes in object names, wrong role specifications or a wrong order of messages to send are easily made. There is no static validation on protocol constraints, and when a protocol becomes more complex with more roles and messages, it is a tedious task to program the business logic without errors. When for example Alice starts waiting before a RES object is sent, the protocol hangs. In this example this is relatively easy to reason about, but with more roles and messages complexity increases, and debugging will become a time consuming task for a programmer when the distributed application is implemented as in this example. Also a check for all roles of the protocol being active, is missing. As a result the roles of the protocol have to be started in the right order as described by the protocol. In this case it is important to start Bob first, since the protocol of Bob and Alice starts with a message from Alice to Bob. When Alice first starts the protocol, the http server of Bob is not ready for receiving the messages from Alice, and messages will be lost.

Alice and Bob illustrate an easy example of a distributed application. Because of this easiness mistakes are easily solved. For example the reasoning that Bob has to start before Alice, because Bob starts listening, and Alice starts with sending. Also, the definition of all the required physical details in the beginning of the program is much easier with an example with only two roles. But as more roles join the party, the harder it becomes. In these

two code examples Alice and Bob can send each message whenever they want, which will also break the protocol. When for example two roles both start waiting on each other, a deadlock situation follows. There is no validation for the order of the messages, the software engineer will have to reason all the flows, and implement, debug and correct these by hand. There is no validation for the correctness of the messages, this will have to be implemented by the software engineer by hand. There is no validation of the details of the involved roles of the protocol. There is no validation for the presence of the roles during the runtime of the protocol, the order of the startup of the roles will also have to be managed and executed by hand. This will probably become a responsibility of the distributed application user. Also the computation logic from the startProtocol methods contain code related to the coordination concern of the distributed application. This involves low level connection details, parsing of messages and http server closing code. This easy Alice and Bob example could have been even more easier when the computation and coordination were separated.

Three conceptual layers with corresponding concrete programming abstractions for distributed applications are discussed in Sections 3.2, 3.3 and 3.4. In Section 3.2 and 3.3 the Alice and Bob example is implemented with the facilities that are present in the corresponding conceptual layer. These code example show the difference between the layers and also the power of the introduced solution. Section 3.4 shows the global description of the Alice and Bob example protocol that serves as the base for the protocol facilities.

3.2. LAYER ONE, ROLE BASED COMMUNICATION

Abstraction layer 1, the role based communication, supplies an abstraction layer directly on the low level tcp/ip facilities of sending and receiving of messages. For the distributed communication between the roles, messages are used. Roles can receive messages from other roles and messages can be sent to other roles. Of course all the roles need the physical Internet address details of the involved roles in the protocol. This way messages can be received by physically separated distributed roles. Also sending and receiving messages is only possible when roles are actively listening for messages. Otherwise messages will be lost, and the communication constraints of the protocol cannot be validated. Also messages can be received out of order, for example due to timing differences between different roles and the used physical networks. An Internet connection between a role that acts via a satellite or a role that communicates via optical fiber will probably result in significantly different timing of the message responses. Therefore messages have to be stored immediately on arrival and will be retrieved when a role needs the specific message. This cannot be a simple sequential queuing mechanism, but a more advanced message database is needed. Storage and selective retrieval of messages is required. The arrival of messages is uncontrollable and can happen continuously, while the receipt of a message when a role needs it is controllable for the receiving role. For a successful first abstraction layer several facilities are essential. Facilities for role based sending, for the arrival of messages and the receiving of messages are needed. But also a warranty for every role to be active when the protocol starts is essential.

In Section 3.2.1 the mediator role is discussed. This system mediator role supplies roles involved in the protocol with physical connection information of other roles. Also the mediator guarantees that all roles will be active when the protocol starts. In Section 3.2.2 the message store is explained. This facility makes storage and selective retrieval of messages

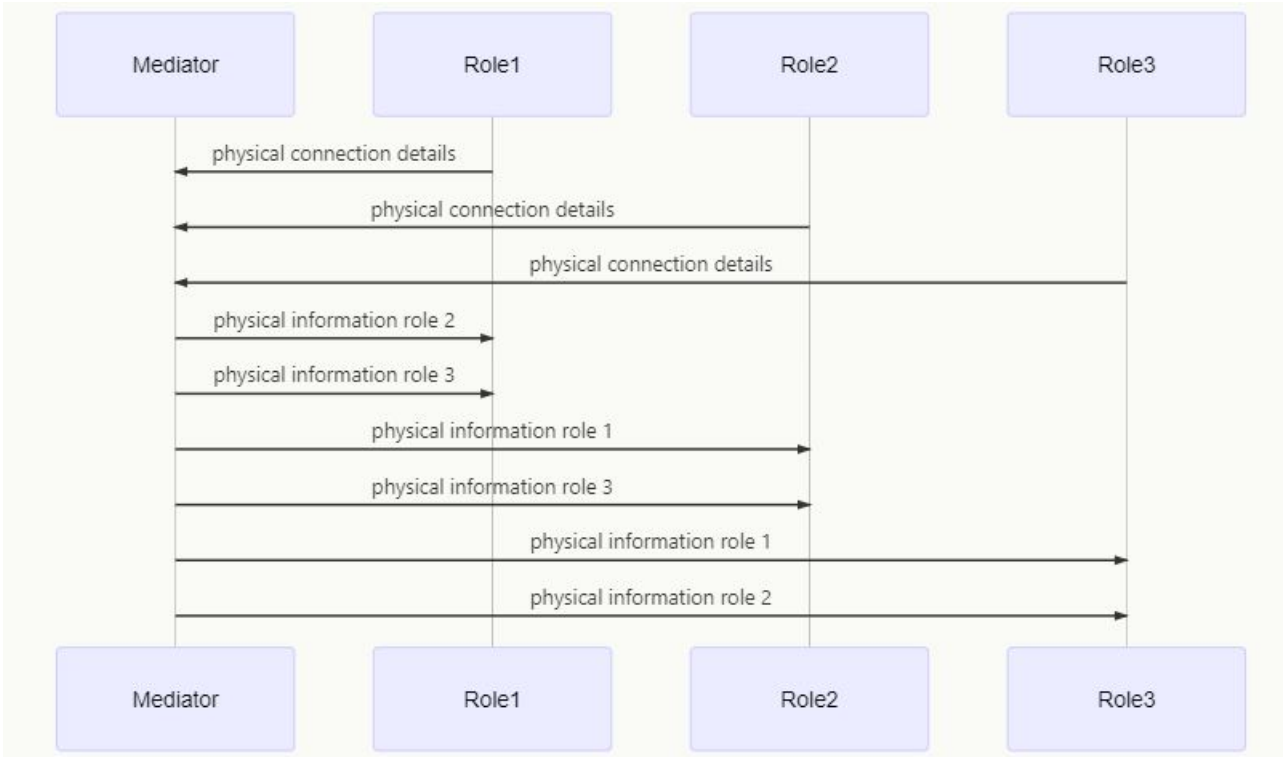

Figure 3.3: Mediator role responsible for protocol startup

possible. Section 3.2.3 describes the role based sending and receiving of messages with an example implementation of Alice and Bob.

3.2.1. MEDIATOR

For a protocol to start, all the roles involved in the protocol should be actively listening at the same time. This way messages that are sent will be received, since all roles are active. This requirement that all roles involved in the protocol have to be active will be enforced with a standard mediator role. This mediator role is a software process that is autonomously active and always has to be started first. All protocols need a mediator for starting up. All custom roles in a protocol will send their connection information to this mediator. When all the physical connection details from the different roles of the protocol are received by the mediator, then the mediator sends the connection details of the different involved protocol roles to all the roles from the protocol. This way all the roles in the protocol know the connection information of all the participant roles in the protocol. When a role has received physical connection details of all the involved roles, the protocol will be started for the role. This way all the roles will be listening actively for messages, and no messages will be lost when a role sends a message. When the protocol has started up, the mediator stops executing and ends its work. The mediator role with the corresponding startup process is illustrated in Figure 3.3. An implementation of the mediator role in TypeScript is shown in code example 3.3.

Listing 3.3: TypeScript implementation of the mediator role

```
import {receiveMessageServer} from './receiveMessageServer';
import {ROLEMESSAGE} from './Message';
import {sendMessage} from './sendMessage';
import {roles, connectedRoles} from './globalObjects';
import {messageDB} from './messageDB';

async function initProtocolByMediator(){
  while ( true ) {
    const msg = await messageDB.remove((m)=>m.name===ROLEMESSAGE.name);
    connectedRoles.save(msg);
    const allRoles = Object.values(roles);
    const missingRoles = allRoles.filter(
      (role) => connectedRoles.missing(role)
    );

    missingRoles.forEach(
      (role) => console.log(`${role} has not started`)
    );

    if ( missingRoles.length > 0){
      continue;
    }

    const relevantRoles = allRoles.filter(
      (role) => role !== roles.mediator
    );

    for ( let i=0; i<relevantRoles.length; i++ ) {
      for ( let j=0; j<relevantRoles.length; j++ ) {
        if (i!=j) {
          const rm = new ROLEMESSAGE(relevantRoles[j]);
          sendMessage( roles.mediator, relevantRoles[i], rm );
        }
      }
    }

    receiveMessageServer.terminate();
    console.log(`protocol started ${new Date()}`);
    break;
  }
}

receiveMessageServer.start(connectedRoles.getInfo(roles.mediator).port);
initProtocolByMediator();
```

The code starts with imports for sending and receiving messages, the complete code of these imported objects can be found in Appendix A. Also a global object that has all involved roles from the protocol is imported with a collection for saving the information of the connected roles. The mediator role starts a local server to be able to receive messages,

which is done in line 43. Then the function initProtocolByMediator is called, which executes an infinite loop and starts waiting for a ROLEMESSAGE.

Listing 3.4: The system ROLEMESSAGE

```
export class ROLEMESSAGE extends Message {
    public host:string;
    public port:number;
    constructor(public roleName:roles){
        super(ROLEMESSAGE.name);
        this.host = connectedRoles.getInfo(roleName).host;
        this.port = connectedRoles.getInfo(roleName).port;
    }
}
```

A ROLEMESSAGE is a predefined system message for sending physical connection information. The implementation is show in Listing 3.4. This message has physical connection details from a role filled by using a internal map with physical connection role details. When mediator role has received a ROLEMESSAGE from every involved role, the collected physical information is sent to all the involved roles. This way the roles know when to start and where to send messages to. Internally for every involved role the connectedRoles map is filled with the physical connection information of all the involved roles.

3.2.2. MESSAGES STORE

A protocol between distributed software processes will easily involve timing issues in the sending and receiving of messages. Some processes will act faster or slower than others. This can have numerous of reasons. The arrival of messages is an uncontrollable process for a role and can happen continuously. The receive of messages is a controllable action of a role, meaning that a message is fetched when a role needs it. To deal with a protocol constraint like the order of messages, messages will have to be saved on arrival. But the message will be fetched when it is needed following the constrained order sequence declared in the distributed protocol definition.

For example, a distributed role process named A has to actively listen for messages from all the involved roles in the protocol. It is possible a message is received from another role called B, but the current state of the local protocol of A first has to wait for a message from role C. The current arrived message from B is needed on a later time, which can happen when B is faster than C. For this reason messages are stored on arrival and fetched on the moment they are needed.

A message store provides a facility for saving the messages on arrival, and functionality for retrieving the messages on a controlled moment. Retrieving messages is based on the type of the message and the role that send the message.

In Listing 3.3 on line five the last import is the message database. The complete implementation of the message database can be found in Appendix A.4. The public API of the message database is shown in Listing 3.5. With the use of the add function messages can be added to the database. By using remove, a message is retrieved and removed from the database.

Listing 3.5: TypeScript Message database

```
interface MessageDB {
    add(msg: Message): void;
    remove(predicate: (msg: Message) => boolean): Promise<Message>;
}
```

The message database is a facility for saving the communication messages of the protocol, and retrieving messages with the possibility to use predicates that define the message type or the role that has sent the message. An example of such a predicate is shown on line nine of Listing 3.6. It is possible to add messages, which is done in the local server of each role; see the implementation of the receive Message server in Appendix A.1. An example of an addition af an ADD message to the message store is shown in code example 3.6 on line five.

Listing 3.6: TypeScript Message database method examples

```
// import the message database, every role has its own
import {messageDB} from './messageDB';

// messages are added with following call
messageDB.add(new BYE());

// a message can be retrieved with following code
const msg = await messageDB.remove(
  (m)=>m.name===RES.name && m.from === roles.bob
);
```

3.2.3. ALICE AND BOB IMPLEMENTED WITH LAYER ONE FACILITIES

The implementation of the Alice and Bob example with the facilities provided by the layer one abstractions is shown in Listings 3.7 and 3.8. Both these code examples start with the import of objects from five different files. The first import of the object receiveMessageServer is for the message server that is responsible for the continuous arrival and storage of the messages. The code of this message server can be found in Appendix A.1. The second import is responsible for the different custom messages. The ADD, BYE and RES messages are imported from the Message library. The implementation of this library can be found in Appendix A.5. The sendMessage method imported on line three is needed for sending messages, the implementation can be found in Appendix A.2. A specification of the different roles from the protocol and a method to initialize the protocol are imported from the globalObjects library. The code from this globalObjects library can be found in Appendix A.3. The initialize method triggers the system Mediator role discussed in Section 3.2.1. The fifth and last import on line five is the message store. This message store is discussed in Section 3.2.2. The complete code of the message store can be found in Appendix A.4.

Listing 3.7: Computation logic of Alice with layer one facilities

```
import { receiveMessageServer } from "./receiveMessageServer";
import { ADD, BYE, RES } from "./Message";
```

```
import { sendMessage } from "./sendMessage";
import { roles, initialize } from "./globalObjects";
import { messageDB } from "./messageDB";

async function executeProtocol(role:roles,port:number,host:string) {
  await initialize(role,port,host);
  for(let i=0;i<5;i++) {
    const add = new ADD(4,2);
    await sendMessage(roles.alice, roles.bob, add);
    const msg = <RES> await messageDB.remove(
         m => (m.name === RES.name && m.from === roles.bob)
    );
    if (msg) console.log(`RES with ${msg.sum}.`);
  }
  await sendMessage(roles.alice, roles.bob, new BYE() );
  receiveMessageServer.terminate();
}
executeProtocol(roles.alice,30001,'localhost');
```

In Listing 3.7 the implementation of the computation logic of Alice in the Alice and Bob example is shown.

The method executeProtocol contains the computation logic and this method is started on line twenty. In this executeProtocol method on line eight the initialize function triggers the Mediator, and starts waiting until the Mediator resolves his Promise. Now the physical details of all the protocol involved roles are present for Alice. On line nine the code executes a loop five times and sends an ADD message with the values four and two to Bob. On line eleven the sendMessage is used; the first parameter of this method is the role from whom the message is sent. The second parameter is the role to whom the message is sent, and the third parameter contains the message itself. On line twelve a message is fetched from the message store. This message has to be from Bob and also have the RES type. On line fifteen the contents of the RES message is displayed. The code ends with the sending of a BYE message to Bob on line seventeen; after that the message server is terminated on line eighteen.

With the facilities of layer one it is easier to specify messages to roles without knowing the physical details of the involved roles. Also the execution order of the Alice or Bob role is not relevant any more, since the Mediator role will have the responsibility for starting up the protocol. Also messages will always arrive with this implementation and fetched until they are needed, this is realized with the facilities from the message store.

But with only the facilities of layer one protocol constraints are not validated. For example a deadlock situation is easily introduced in the code. Messages can be sent whenever a role wants, and validating the right message flows will have to be done by the software engineer who implements the distributed application. There are no static validated protocol constraints in this layer, only the facilities for easily sending messages to involved roles. Mistakes with physical details, the startup of the protocol, incorrect messages are reduced. But errors like sending messages that a role is not supposed to send, or for example trying to fetch a message that will never arrive are mistakes that are easily made with only the facilities of layer one. For sending and receiving messages based on roles, roles and messages have to be defined. These are essential elements in a distributed protocol: before the

protocol starts the possible messages are known, as well as the involved roles.

This role based sending and receiving facility for the first abstraction layer will provide a way to send a message to a role by only providing the name of the role with the message. The physical low level tcp/ip details of the roles are not needed at this moment; this information was already initialized by the mediator and present for the active role. This layer one will help hiding the distributed communication low-level details and facilitate the starting up of the protocol, but admissible protocol flows are not enforced. The running code can easily result in a deadlock situation, certainly when more roles join the party. With a simple protocol like 'Alice and Bob', this is not instantly the case, and the programmer is able to oversee the deadlock situations. But with more roles it is hard to oversee all the different situations in the protocol; an example of the rapid increase in complexity is given in Section 3.5.2.

Listing 3.8: Computation logic of Bob with layer one facilities

```
import { receiveMessageServer } from "./receiveMessageServer";
import { ADD, BYE, RES, Message } from "./Message";
import { sendMessage } from "./sendMessage";
import { roles, initialize } from "./globalObjects";
import { messageDB } from "./messageDB";

async function executeProtocol( role:roles, port:number, host:string) {
  await initialize(role, port, host);
  while ( true ){
    const msg = await messageDB.remove(
         m => (m.name === ADD.name && m.from === roles.alice)
           || (m.name === BYE.name && m.from === roles.alice)
    );
    switch (msg.name) {
      case ADD.name: {
        const add = <ADD> msg;
        console.log(`ADD with ${add.value1} and ${add.value2}.`);
        const res = new RES( add.value1 + add.value2 );
        await sendMessage(roles.bob, roles.alice, res);
        break;
      }
      case BYE.name:{
        receiveMessageServer.terminate();
        return new Promise( resolve => resolve() );
      }
    }
  }
}
executeProtocol(roles.bob,30002,'localhost');
```

Code Listing 3.8 has the implementation of Bob and only uses the facilities of layer one. The executeProtocol method is executed on line twenty nine. This method will first trigger and wait for the system Mediator role on line eight. Then in an infinite loop Bob starts fetching available messages on line ten. The criteria for the messages that are fetched are,

a BYE from Alice or an ADD from Alice. These used message criteria can be seen on line eleven and twelve. When an ADD is received, the message is casted to an ADD message on line sixteen, and serves as an input for the RES message on line eighteen. This RES message is sent back to Alice on line nineteen. After the receive of a BYE message, the message server is stopped on line twenty three and the executeProtocol method returns a fulfilled promise ending the protocol.

3.3. LAYER TWO, STATIC PROTOCOL VALIDATION

The second abstraction layer provides an API with which the distributed application can be programmed following the constraints of a software protocol definition. Only admissible message passing is possible, and extra help programming the distributed application is achieved with code navigation and automatic code completion facilities. For generating an API, a global protocol is translated to state machines for every role present in the protocol. The foundation for this layer is a description of the global protocol in a JSON data file, in this file the different state machines for each roles are described. In Section 3.3.1 the local state machines and the corresponding STD in a protocol are discussed. In Section 3.3.2 the foundation for generating an API for protocol programming is discussed. This involves a global protocol description in JSON format. This global JSON definition exists of a specification of the local role definitons in one file. The last section contains additional comments about the second abstraction layer, this will be Section 3.3.4.

3.3.1. LOCAL PERSPECTIVE STATE MACHINES

In the Alice and Bob protocol from example (1) there are two roles, which means there are also two local perspectives with their corresponding state machines. The STD diagrams, used to visualize the state machines of the local role perspectives, are presented in Chapter 2.1.3. Figure 2.3 shows the perspective of Alice and Figure 2.4 shows the perspective of Bob. To reduce pageturns the two STD diagrams are shown again in Figure 3.4.

The STD of Bob has three states, one initial state, one final state and a normal state. This is also the case with the diagram of Alice. The initial state is S1 and has a thick line, the final state is S3 and has a double line and is solid filled. The transitions represent the sending or the receiving of the messages. Alice can send a BYE or an ADD message when she is in state S1. When Alice sends an ADD message, she enters state S2. In S2 Alice must wait for a receive of a RES message. When Bob is in the initial state S1, he can receive an ADD message or a BYE message. When Bob receives an ADD message he enters the state S2 from his local perspective, after this he sends back a RES message and comes back in state S1.

State machines can be represented as APIs in TypeScript as follows. The states of the state machines are translated to private TS classes and public TS interfaces. The transitions between the states are methods in the corresponding state classes and interfaces. When a transition is made, a method of a TS object that represents a state is called. This method executes the transition and returns the next state object.

Alice will have two methods in the object that represents the initial state: one for sending an ADD message and another for sending BYE. The object that represents Alice's second state will have a method to wait for a RES message. The final states will not have any methods, since no transitions can be made from the final state.

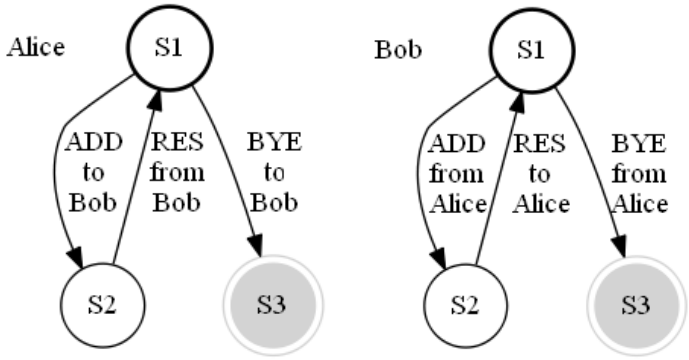

Figure 3.4: Local perspective of Alice and Bob

The interface of Bob should have a method that waits for a BYE or an ADD message. After receiving the ADD Bob should have a possibility to send a RES back to Alice, according to the protocol description. This RES contains the result of the executed business logic, in this case the summation of two numbers. The generated API should offer these methods when they are needed and possible, according the description of the protocol.

The result is an API for the software engineer for help with implementing the business requirements: there is no more concern about message passing, this is dealt with the framework. The required workflow between the distributed processes is defined in a higher level protocol description. Abstraction layer three deals with this higher level description of the global protocol; this is explained in Section 3.4. The resulting protocol defined admissible order of sending and receiving messages will be statically validated. The generator generates an API for every role. With this interface messages can be received from other roles and messages can be sent only on the correct times. The methods of the objects in this interface return other objects that correspond with the successor states in the protocol. The protocol is internally translated to a state transition diagram STD for each role. The transitions of the STD illustrate the sending or receiving of messages, and the states correspond to the place where the specific role is at the moment in the protocol.

Listing 3.9: Method signature for protocol computation logic

```
1   f: (start: Alice_Start) => Promise<Alice_End>
```

Every role in a protocol is internally represented as STD, and has one initial state and one final state. A method with the initial state as parameter and the final state as a result will be the signature for the method containing the implementation of the custom computation logic.

The TS datatype definition for the function to contain the computation logic is shown in Listing 3.9. The protocol business logic is implemented in a function that has an initial state of the local perspective as input parameter, in the example the object Alice_Start, and the result of the function will be a final state of the internal state machine of the role, in the example Alice_End. The Promise object is a way in JavaScript to return and wait on a future asynchronous result. The Alice_End state is returned in a Promise, this forces the API to complete the state machine.

3.3.2. JSON WITH LOCAL PERSPECTIVES

The foundation for generating a protocol API for distributed communication is a description of the global protocol in a specific JSON data format. The FSM from the local perspectives of the protocol participants are shown in the state transition diagram in Figures 2.3 and 2.4 from Chapter 2.1.3. These two diagrams represent the local perspectives of the protocol from the 'Alice and Bob' example (1). The FSM from the roles of the protocol, are used to generate an API for the end user, the software engineer. States will be generated to objects, from which transitions can be done to other states by sending or receiving a message. For generating this protocol API a textual representation of the protocol with the local FSM of the roles in JSON data format is used.

Listing 3.10: Alice and Bob protocol description in JSON format

```
{ "roles": ["Bob","Alice"],
  "protocol": [
    { "role": "Bob",
    "states": [ { "name": "S1",
                  "type": "initial",
           "transitions": [ {"op"      : "recv"
                            ,"message" : "Add"
                            ,"role"    : "Alice"
                            ,"next"    : "S2"},
                            {"op"      : "recv"
                            ,"message" : "Bye"
                            ,"role"    : "Alice"
                            ,"next"    : "S3"} ] },
                { "name": "S2",
                  "type": "normal",
           "transitions": [ {"op"      : "send"
                            ,"message" : "Res"
                            ,"role"    : "Alice"
                            ,"next"    : "S1"} ] },
                { "name": "S3",
                  "type": "final",
           "transitions": [] } ] },
    { "role": "Alice",
    "states": [ { "name": "S1",
                  "type": "initial",
           "transitions": [ {"op"      : "send"
                            ,"message" : "Add"
                            ,"role"    : "Bob"
                            ,"next"    : "S2"},
                            {"op"      : "send"
                            ,"message" : "Bye"
                            ,"role"    : "Bob"
                            ,"next"    : "S3"} ] },
                { "name": "S2",
                  "type": "normal",
           "transitions": [ {"op"      : "recv"
```

```
37                              ,"message" : "Res"
                                ,"role"    : "Bob"
39                              ,"next"    : "S1"} ] },
                  { "name": "S3",
41                  "type": "final",
              "transitions": [] } ] }
43    ]
}
```

JSON Listing 3.10 shows the local protocol descriptions based on the 'Alice and Bob' example (1). This input JSON is used by the protocol generator and consists of a local protocol definition per role. The roles array summarizes the required participants of the protocol. The protocol array describes the local perspectives of the different roles. The objects of the protocol array consist of the name of the role and the states that are present in the state transition diagram of the corresponding local protocol. The states object is an array of state objects, that have the name of the state, the state type and the transitions that can be made from the state to other states. These transitions can be done by sending or receiving messages. There are three state types, namely initial which starts the STD, a normal state type for the regular states and final state type, which is the state where the STD from the role ends. A restriction on the input JSON data protocol is that there can be only one initial state and only one final state for one local perspective of a role. The transitions data field is an array with transitions that are made from a specific state. A message receiving or sending will always trigger a transition. This transitions array has objects that represent a single transition. The message field represents the message that is received or sent in the transition. The op field in this object can be of the type send or recv, which means that a message is send or received in this specific transition. The role is the participant where the message will be sent to, or from which the message is received. The next field represents the state that follows after the transition is taken.

3.3.3. ROLE IMPLEMENTATION IN TYPESCRIPT

The generation of an API based on the description of the protocol in the JSON from Listing 3.10 results in static validated protocol API for all roles involved in the protocol, in this case Alice and Bob. In Listing 3.11 the public exported objects of the API that is generated for the role Alice are shown. The complete generated API for Alice can be found in Appendix A.6.

Listing 3.11: Generated API for Alice

```
interface IAlice { }
2 interface IAlice_S1 extends IAlice {
      readonly messageFrom: roles.bob;
4     readonly messageType: messages.RES;
      message?: RES;
6     send_ADD_to_Bob(add: ADD): Promise<IAlice_S2>;
      send_BYE_to_Bob(bye: BYE): Promise<IAlice_S3>;
8 }
interface IAlice_S2 extends IAlice {
10    recv(): Promise<IAlice_S1>;
```

```
}
interface IAlice_S3 extends IAlice { }

type Alice_Start = IAlice_S1;
type Alice_End = IAlice_S3;

async function executeProtocol(
    f: (Alice_Start: Alice_Start) => Promise<Alice_End>
    , host: string
    , port: number ) {
  await initialize(roles.alice, port, host);
  let done = await f(new Alice_S1());
  return new Promise<Alice_End>(resolve => resolve(done));
}
```

The complete generated code for Bob can be found in Appendix A.7. The interfaces IAlice_S1, IAlice_S2, IAlice_S3 are the objects that represent the states from the STD of the local pespectives from the role Alice. The start state gets the synonym Alice_Start and the final state receives the synonym Alice_End. The function executeProtocol has the function f as parameter: this is the function with the computation logic. The TS datatype definition of this function f is shown in Listing 3.9. This method starts with an initialize on line 21 in Listing 3.11. During this initialisation the local physical settings are sent to the mediator, and Alice starts to wait until the mediator sends back the information of the other roles. This is done in the method initialize. The implementation of the initialize function can be found in Appendix A.3. After the initialisation an instance of the start state of Alice is created and passed to the parameter f, which is a function. The f function will be supplied by the software engineer who is responsible for implementing the computation logic of the distributed application. The executeProtocol function then waits for a final state to return, that is passed back as a promise, therebye promising the protocol will end.

Listing 3.12: Computation logic for Alice using the generated API for Alice

```
import {Alice_Start,Alice_End,executeProtocol} from './Alice';
import {ADD,BYE} from './Message';

async function protocol(s1:Alice_Start):Promise<Alice_End> {
   for(let i=0;i<5;i++) {
      const add = new ADD(21,21);
      const s2  = await s1.send_ADD_to_Bob(add);
      s1 = await s2.recv();
      if (s1.message)
        console.log(`Received ${s1.message.sum}`);
   }
   const done=s1.send_BYE_to_Bob(new BYE());
   return new Promise( resolve => resolve( done ) );
}

async function start(){
   await executeProtocol(protocol,'localhost',30001);
```

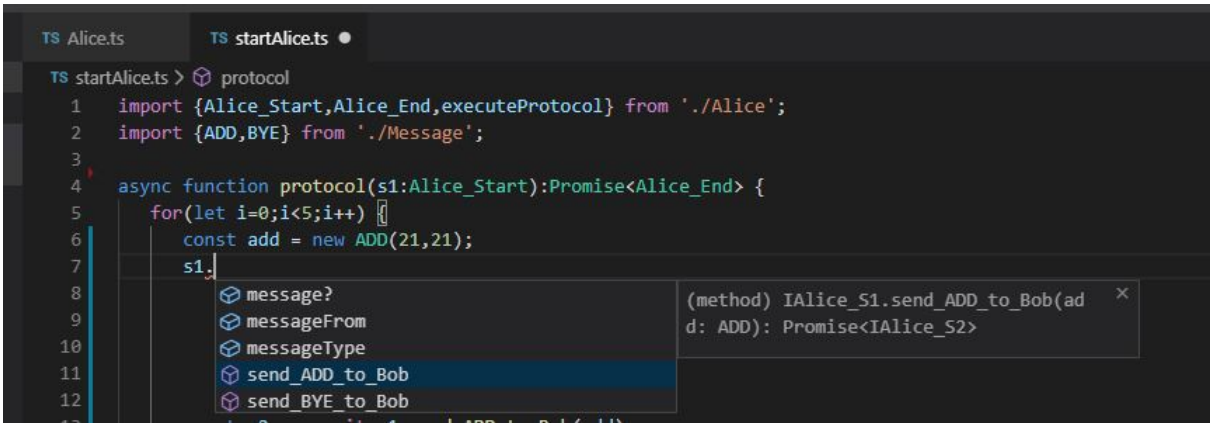

Figure 3.5: Protocol statically checked and statement completion

```
18 }

20 start();
```

A possible implemention of the business logic for Alice is shown in code example 3.12. The first line imports three objects from the generated interface from Alice. The intial state object called Alice_Start and the final state object called Alice_End are imported. Also the method executeProtocol is imported, which is used to pass the computation logic to the coordination logic of the protocol. The function named start on line twenty starts the business logic for Alice. In the start method the executeProtocol function from the role API from Alice is called. The business logic function named protocol from line four is passed as parameter to the executeProtocol function. And also the physical connection details of Alice are passed as parameters. The method named protocol has the implementation of the computation logic of Alice, in this case five times sending the two numbers to Bob, and waiting for the result, and then displaying the information. After the five ADD messages a BYE is sent, and the protocol ends. The s1 object of the type Alice_S1 has two methods, one for sending an ADD and another for sending a BYE. After sending an ADD, the object returned from the method send_ADD_to_Bob on line seven returns an object that has a method for waiting on a RES. This object is saved in the s2 variable. This way admissible flows of the protocol are enforced. It is not possibe to send two ADD messages right after each other, since the protol specification does not allow it. This is checked at runtime, since the type system in TypeScript does not have such a check during static compilation.

In Figure 3.5 an example is illustrated in which the result of explicitely stating the data type of the s1 parameter from the protocol function is shown. The benefit of a statically validated distributed software protocol is not only the protocol error checking, but with the use of a decent IDE facilities for guiding the distributed programming process are also available. These facilities include code completion, code navigation and even automated code refactoring.

Figure 3.6 show an example of data type inference, which is available in the TS language. Data type inference means the compiler is capable of determining the data type itself. In this case the variable s2 is declared the right state data type. This is done implicitely by the compiler. This way the STD does not have to be memorized by the programmer; instead,

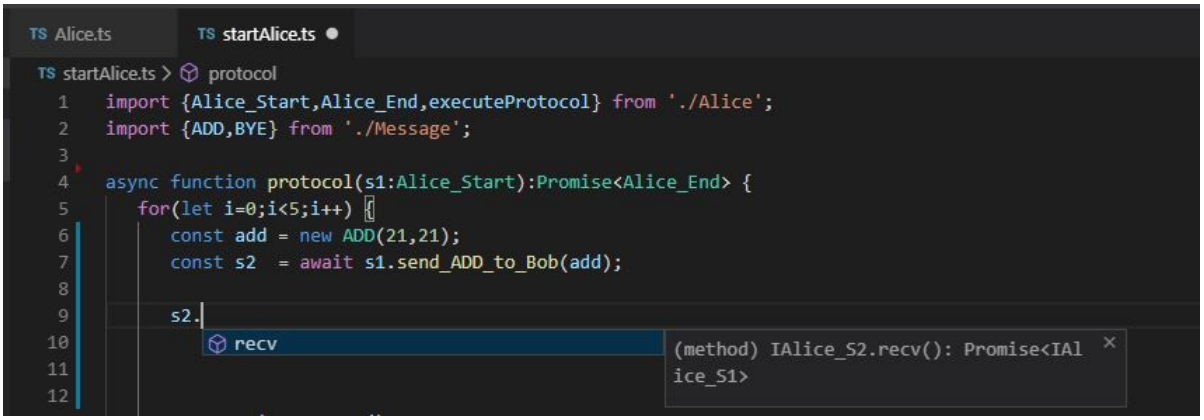

Figure 3.6: Protocol data type inference

the types returned from the transition methods are inferred. As a result the transitions that are available in a next state will be available with statement completion.

Listing 3.13: Computation logic for Bob using the generated API for Bob

```
import {Bob_Start,Bob_End,executeProtocol, messages} from './Bob';
import {RES} from './Message';

async function protocol(s1:Bob_Start):Promise<Bob_End> {
  let nextState = await s1.recv();
  while ( true ){
    switch (nextState.messageType) {
      case messages.ADD: {
        console.log(`Received ${nextState.message.value1} and ${nextState.message.value2}`);
        const res = new RES( nextState.message.value1 + nextState.message.value2 );
        s1 = await nextState.send_RES_to_Alice(res);
        nextState = await s1.recv();
        break;
      }
      case messages.BYE:{
        const done = nextState;
        return new Promise( resolve => resolve(done) );
      }
    }
  }
}

async function start(){
   await executeProtocol(protocol,'localhost',30002);
}

start();
```

Code example 3.13 shows the implementation of the computation logic for Bob. The

code start with an import of the public generated role API for Bob on line one. The function protocol defined on line four has the implementation of the computation logic. This computation logic is passed to the coordination logic on line twenty four, thereby seperating the computation and coordination concern.

3.3.4. ADDITIONAL COMMENTS

The generated API for the distributed protocol guarantees the liveness property, all the messages that are supposed to be sent, will be send. And also the safety property is guarantied: messages that should not be send, cannot be sent.

Also this layer separates computation from coordination; separating these concerns will increase code maintainability. In Chapter 1.3 was already mentioned the relevance of serarating concerns in programming code. In the implementation of the compution code of Alice, in Listing 3.12, the computation logic from Alice is passed to the coordination logic from the protocol. The method protocol is passed as a parameter to the imported function executeProtocol. This is the intended way to work using these abstraction layers to implement the distributed application. The implementation of Bob, in Listing 3.13, also illustrates this principle.

The problem with this layer is that writing this data JSON source, which is the basis for generating the protocol API, can be very tedious. A mistake is easily made especially when more roles join the party. This will increase the complexity of the involved statemachines rapidly. This makes this layer not sufficient enough to offer to a software engineer who is responsible for the implementation of a distributed application.

3.4. LAYER THREE, GLOBAL PROTOCOL DEFINITION

A global description of a protocol, preferable in a intuitive language like Scribble, is the ideal syntax format for describing distributed interactions between processes. Describing the global protocol instead of describing each role perspective separately provides a higher abstraction layer to the software engineer. The local perspectives can and should be be generated from the global perspective. This global protocol description is less work for the software engineer and provides a higher abstraction view of the software communication protocol. After all, instead of looking at different local protocol descriptions separately it is easier to look at and focus on one global description that describes the complete software communication protocol.

To be able to use the static validation facilities of layer two a JSON in the right format, that describes the global protocol, is used to generate the protocol API. Therefor a parser is needed for the parsing of a global protocol description to the JSON data format. This parser is beyond the scope of this project. But with help from unpublished research in the field of parallel programming, an evolved Scribble parser is used for this research. This parser is capable of parsing an evolved Scribble syntax and generate the needed JSON data file for abstraction layer two.

Therefore, for this project a Scribble look al like syntax is used for globally describing example protocols. The syntax of this language looks like Scribble language and provides higher level language constructions that are easy and sufficient for describing distributed protocols on a global level.

In Listing 3.14 an example of a global protocol description of the Alice and Bob example

(1) is shown.

Listing 3.14: Global protocol definition of Alice and Bob

```
module AliceBob;
global protocol AliceBob(role Alice, role Bob) {
    choice {
        ADD from Alice to Bob;
        RES from Bob to Alice;
        do AliceBob(Alice, Bob);
    } or {
        BYE from Alice to Bob;
    }
}
```

The Scribble syntax used in the Alice and Bob example (1) from Chapter 2.1.3 in Listing 2.1 is different from the syntax used in Listing 3.14. The code in Listing 3.14 starts with a declaration of the module. The syntax from Listing 3.14 is more compact than the listing from Chapter 2.1.3 and eliminates the need for the message details, in this case the integer payloads. Also the choice keyword can be used without the role specification, since the master during the choice is the role that has the first action after the choice. The specific contents of a message is also not needed; these details can be filled in by software engineer implementing the business logic. This higher lever syntax closely illustrates the sequence diagram from Figure 2.2 from the Alice and Bob example (1).

3.5. CONCLUSION

With a simple example the three proposed abstraction layers for simplifying protocol programming were illustrated in this chapter. The end goal for this master thesis is to simplify role-based communications programming in JavaScript/TypeScript. This will substantiate the idea that role-based communications programming is a good alternative solution for programming constrained distributed applications. Role-based programming can make programming distributed software applications more easy and comprehensive. The lower level protocols like tcp/ip, http, or a rest architecture are no longer relevant for the software engineer who implements the distributed application. A complete focus on implementing the real business logic is the logical result.

This offered framework will generate the code that is responsible for the coordination of the distributed program and also separate the coordination programming code from the code that is responsible for the custom business logic computation. This will result in more maintainable programming code as described in Section 1.3.

Programming abstractions for implementing distributed software communication will probably result in the faster development of distributed applications. The programmers of the future will 'stand on the shoulders of giants' [Tur59]. But now with an eye on new software abstraction layers, on top of others.

Some extra attention for the complete process for generating a protocol API based on the three proposed abstraction layers is recalled in Section 3.5.1. To illustrate the increase in complexity when more roles join the multi-party protocol, the protocol from Alice and Bob is extended with another role. This is described in Section 3.5.2.

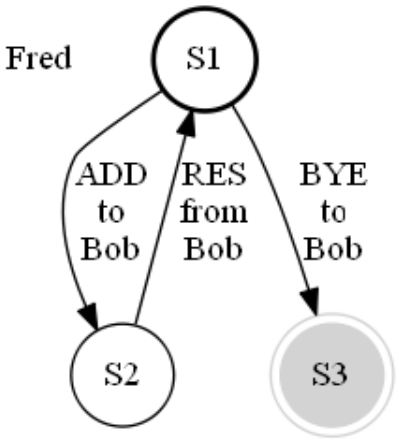

Figure 3.7: Local perspective of Fred

3.5.1. API GENERATING PROCESS

This Section gives a summarized description in a chronological order of the steps that are needed to build a distributed constrained application, based on a description of a global protocol. Several actions are needed for the implementation of a distributed software application using the described method in this chapter.

First a protocol is described on a global level in a Scribble evolved syntax, this code will be parsed and the syntax is validated. This high level format describes the flow of the messages between the roles. After a successful validation of the global protocol description a transformation is executed to a JSON data format protocol definition. With the help of this data, the protocol framework generator constructs an API for every role involved. These interfaces are generated for each role and are based on the statemachines of the corresponding roles. States are objects, and transitions are methods of these objects. A transition represents a situation where a role waits or sends a message to another role.

The custom computation logic can be implemented by the software engineer. The code of the computation logic starts with the initial state of the involved role of the protocol, and ends with the final state of the local protocol of the involved role. The computation code can be implemented separated from the coordination code and passed as an argument to the protocol coordination framework.

Such an API for implementing the details of the software protocol could be applicable and beneficial a for any statically compiled language. There will be an API for every role involved in the described global protocol, and applicable for distributed application development.

3.5.2. COMPLEXITY

The Alice and Bob example is very easy and has only three states per role. To illustrate the high increase in states and therefore complexity another role is added to the Alice and Bob example (1), namely Fred. Fred does the same as Alice: he sends an ADD or a BYE to Bob. After sending an ADD he waits for a result in the shape of a RES message. After sending the BYE message he ends the protocol.

The STD of Fred is the same as the one of Alice and is illustrated in Figure 3.7, but the diagram of Bob changes after adding the Fred role. Bob now has to deal with two participants,

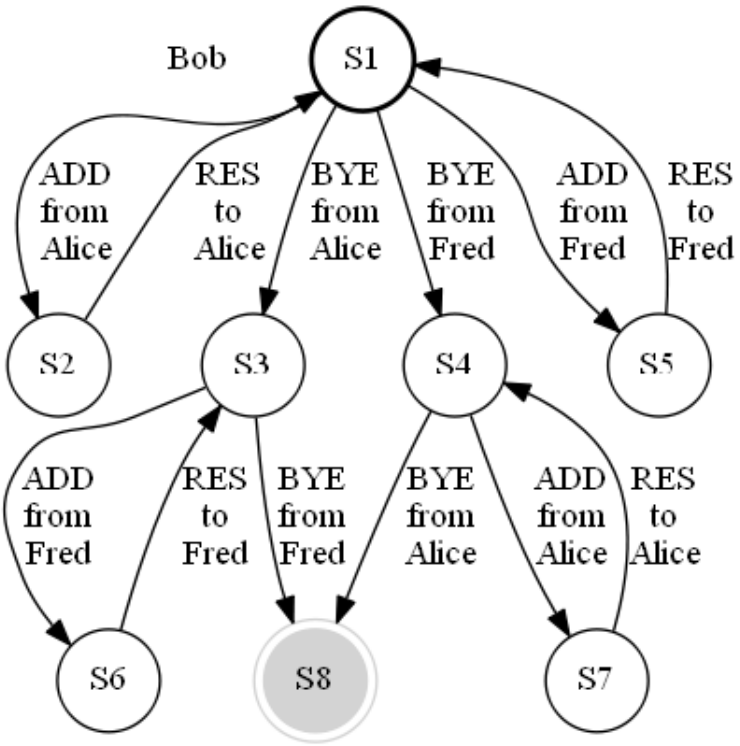

Figure 3.8: State diagram for Bob when dealing with an extra role

for example when Alice or Fred sends a BYE, the protocol does not end for Bob. There still is one role active. The STD of Bob is illustrated in Figure 3.8 and shows eight states instead of three, just after adding one role.

The global description of the protocol also changes, a description is shown in Listing 3.15. The global protocol has four choice options now, and after Alice or Fred stops the protocol becomes the same as the 'Alice and Bob' example.

Listing 3.15: Scribble implementation of Alice and Bob

```
module AliceBobFred;

global protocol AliceBobFred(role Alice, role Bob, role Fred) {
    choice {
        ADD from Alice to Bob;
        RES from Bob to Alice;
        do AliceBobFred(Alice, Bob, Fred);
    } or {
        ADD from Fred to Bob;
        RES from Bob to Fred;
        do AliceBobFred(Alice, Bob, Fred);
    } or {
        BYE from Alice to Bob;
        do Two(Fred, Bob);
    } or {
```

```
16          BYE from Fred to Bob;
            do Two(Alice, Bob);
18      }
    }
20
    aux global protocol Two(role Client, role Server) {
22      choice {
            ADD from Client to Server;
24          RES from Server to Client;
            do Two(Client, Server);
26      } or {
            BYE from Client to Server;
28      }
    }
```

The corresponding sequence diagram of this Alice, Bob and Fred example is shown in Figure 3.9.

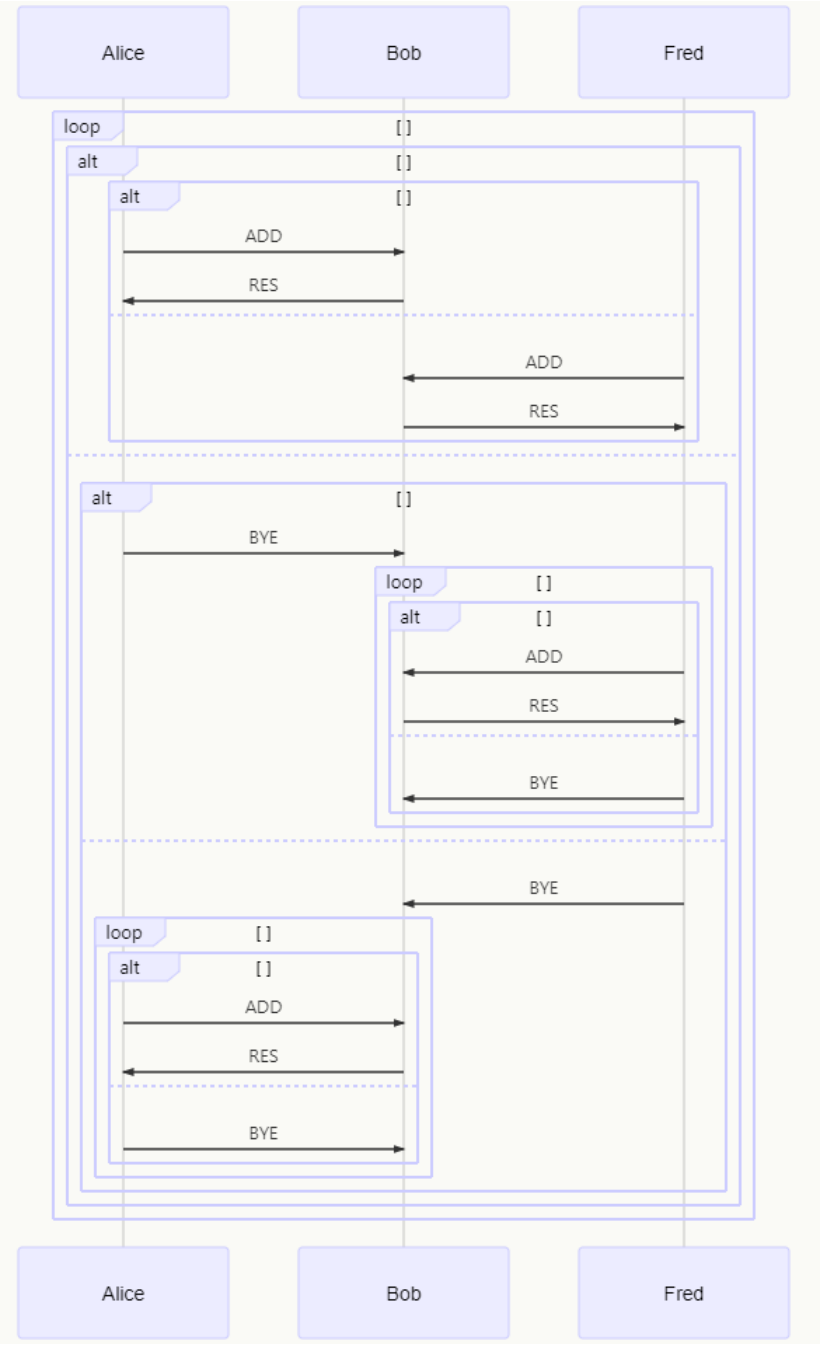

Figure 3.9: Sequence diagram of Alice, Bob and Fred example

4

CONCLUSION

Research for finding the answers to the research questions has been conducted. During this research concrete protocol abstractions have been developed. Several example protocols have been formulated and tested, this way validating the proposed solution from Chapter 3. As a result the research questions can now be answered.

The main research question of this master project is: *How can protocol abstractions be added to TypeScript?*

And this research question is narrowed down in the following five subquestions.

- RQ1: What abstractions do we need?
- RQ2: Do we need a protocol engine for validating and execution of protocol instances?
- RQ3: How is message passing implemented?
- RQ4: Is it possible to add these abstractions as GPL or will it better to add them as DSL.
- RQ5: What is the performance penalty of these abstractions?

More details about the research questions can be found in Chapter 1.5.

The answer to the main research question is: protocol abstractions can be added by providing several concrete abstractions with which protocol programming in TypeScript will be possible. These concrete abstractions will also supply an API for programming the distributed application. This API is based on a global protocol definition and delivers static validation of protocol constraints. These concrete abstractions are divided among three conceptual layers that are on top of tpc/ip and http, Figure 3.1.

In this Chapter 4 the first Section 4.1 discusses the first narrowed down research question. Section 4.2 answers the second research question. Next Section 4.3 discusses the third research question. Section 4.4 answers the fourth research question and Section 4.5 discusses the last research question. In Section 4.6 the validation of the project is discussed and more details about the scope of the project can be found in Section 4.7. The last section of this chapter, Section 4.8 will have a final evaluation on the research project.

4.1. RQ1:WHAT ABSTRACTIONS

What are the concrete abstractions that are needed for defining distributed software communication protocols in TypeScript?

To realize statically validate protocol programming in TypeScript several concrete abstractions are needed. With formulating the conceptual layered architecture from Chapter 3 the different concrete abstractions are categorized among the layers.

The first abstraction layer on top of the base tcp/ip layer is also called 'Role based communication'. This first layer has the mediator abstraction for starting up the protocol and supplying all roles with physical details. The mediator role guarantees all roles involved in the protocol are present when the protocol starts. This system mediator role is discussed in Section 3.2.1. Another concrete abstraction in the first layer is the message store, this abstraction provides a controllable way for fetching messages. The message store is discussed in Section 3.2.2. Role based sending of messages, without the need for physical address details, is also a concrete abstraction of layer one. The ability to save messages immediately on arrival in the message store, is another concrete abstraction that layer one provides. More details about layer one are found in Section 3.2.

The second conceptual layer is the 'static protocol validation' layer. This layer provides statically validated protocol constraints at compile time. The details of this layer are discussed in Section 3.3. In short an API for each role is supplied for statically validated protocol programming. This generated API is the concrete abstraction from this second abstraction layer. The second architectural abstraction layer uses the first conceptual layer.

The third conceptual layer is also called 'Global protocol definition'. This layer facilitates in a user friendly way of defining a global protocol. The software engineer responsible for programming the distributed application will formulate the global protocol in higher level syntax. This mainly means defining the message interactions between the different roles, in other words, implementing the coordination between the distributed processes with an appropriate syntax. From this global definition the framework for programming the distributed application is generated. The coordination concern from the software protocol is generated, and the computation concern can be built next. This global protocol definition with its specific syntax is the concrete abstraction of this layer three.

4.2. RQ2:PROTOCOL ENGINE

The second narrowed down research question is: Do we need a protocol engine for validating and execution of protocol instances? Are runtime facilities needed for distributed software communication protocol abstractions in TypeScript? And if a runtime engine is needed, how is this runtime engine implemented?

When the first conceptual layer is present, the role based communication layer, the runtime facility to guarantee the startup of all wanted involved roles is present too. This facility is the system mediator role from abstraction layer one, also explained in Section 3.2.1. Since all roles are distributed, this role is needed to guarantee the presence of all the roles. This is an engine for the execution of the protocol that is started first. As soon as the protocol is started up completely and all distributed roles involved in the protocol are present, this system mediator role stops and ends its processing.

When the second conceptual layer is also present, the static protocol validation layer, the protocol API for the software engineer is available. Static protocol validation is implemented based on state machine simulation, as described in Section 3.3. When a transition between states in a STD is made more than once, the protocol is incorrect. This means when a method of a state object is called more than once, the protocol is incorrect. This cannot be validated at compile time and is therefore implemented as a runtime validation

check.

These two runtime facilities are needed for protocol validation and the execution.

4.3. RQ3:MESSAGE PASSING

How is message passing implemented? When protocol abstractions for distributed software communication are added, how will the technology for passing messages between distributed software processes be implemented?

Messages are fundamental for distributed software communication. The interactions between the roles with corresponding message for each interaction, are described in conceptual layer three. This is done by defining the message that is passed from role to role. This is illustrated in code Listing 4.1, in this code example an ADD message is send from Alice to Bob.

Listing 4.1: Message from role to role

```
1   ADD from Alice to Bob;
```

The details of the ADD message are specified separately. Specifying message details is the responsibility of the software engineer who implements the distributed application. In Listing 4.2 the ADD message is specified with more details. The ADD message contains two number values, namely value1 and value2.

Listing 4.2: Message definitions

```
1  abstract class Message {
       constructor(public name:string){};
3  }

5  export class ADD extends Message {
     constructor(public value1: number, public value2: number) {
7        super(ADD.name);
     }
9  }
```

A complete example of an implementation of a detailed message specification can be found in Appendix A.5. Besides the specification of the messages, there are also other abstractions needed to implement message passing for protocol programming. Facilities for the uncontrolled arrival of messages, the controlled processing of messages and message sending are also needed.

For the continuously arrival of messages a message server is created that saves messages on arrival in a message store. The message store is explained in more detail in Section 3.2.2 and the implementation can be found in Appendix A.4. The implementation of the message server can be found in Appendix A.1.

The processing of messages can be done when a role is ready for the specific message. This depends on the global protocol specification, which led to a state machine based API for each role. When a state object of an API expects a specific type message from a specific role, it can be fetched from the message store. A role will wait for a message when the message has not arrived yet.

Messages can be sent and received with the generated methods from the API from the role. These methods are the transitions of the state machines that are involved. This API is the conceptual layer two abstraction. Conceptual layer one provides a method for sending messages based on a role name, without the need to specify the physical details. These were already initialized by the mediator; the implementation of this method can be found in Appendix A.2.

4.4. RQ4:GPL OR DSL

Is it possible to add protocol abstractions as GPL or will it be better to add them as DSL. What is the most convenient way for adding software communication protocol abstractions to TypeScript?

Since TypeScript is an open source language with its source code on GitHub, it is possible to add the third conceptual layer protocol abstractions directly as keywords to the TypeScript language. The TypeScript compiler could generate the protocol framework to JavaScript.

This Master research project implemented the protocol abstractions as an external DSL that is modeled in the third conceptual layer. The main reason for implementation as a DSL was to guarantee the feasibility of this Master research project.

But also the number and different kind of abstractions that are needed to implement protocol abstractions, could justify a separate DSL for protocol abstractions. Generating code for a distributed application produces a framework of code, which cannot be translated one to one. The TypeScript compiler transpiles TS to functionally equivalent JavaScript. Protocol abstractions involve facilities like a message store and message server. The complexity of these abstractions and total number of minimal needed abstractions justify a separate external DSL. This external DSL for specifying the coordination will be the basis for the framework that will be generated.

When using coordination abstractions created for TypeScript specific, then generating a complete framework for coordination from TypeScript to JavaScript breaks the main reason of TypeScript. This main reason is, adding static typing to JavaScript.

Also for a higher maintainability a good separation of computation code from coordination code is recommended; with a separate external DSL this is guaranteed.

4.5. RQ5:PERFORMANCE PENALTY

The fifth narrowed down research question is: what is the performance penalty of these abstractions? Is there a significant amount of time extra spent on this generated code layer in TypeScript?

For answering this research question an example protocol, called perfect number has been designed. First this example is implemented in a regular way without the use of a generated protocol API. And then a version generated with static protocol validation, using the facilities of all the conceptual layers from Chapter 3. The perfect number example protocol has a map reduce pattern: first one main process divides work to five other processes. Then the executed work is collected and processed by the main process.

The goal is to calculate perfect numbers. A perfect number is positive number that is equal to the sum of its proper divisors. An example is the number six, which has the divisors one, two and three. The summation of these divisors is six, which makes six a

perfect number. Other perfect numbers are 28, 496 and 8128.

The protocol has six roles, one role called p and the roles s1, s2, s3, s4 and s5. The role called p divides numbers among the five other different processes. These processes calculate the sum of divisors from every received number. When the sum of the divisors of the received number is equal to the received number, the number is a perfect number and is send back to role p. P collects the perfect numbers from the different calculating s roles. When all the numbers are sent by role p, p starts waiting for the calculated perfect numbers.

A short impression of the global description of the protocol is given in example 4.3, the complete global protocol can be found in A.8.

Listing 4.3: Global protocol definition of perfect number example

```
module perfectNumber;

global protocol perfectNumber( role p, role s1
          , role s2, role s3, role s4, role s5 ) {
    choice {
        CALC from p to s1;
        do perfectNumber(p,s1,s2,s3,s4,s5);
    } or {
        CALC from p to s2;
        do perfectNumber(p,s1,s2,s3,s4,s5);
    } or ...
    } or {
        BYE from p to s1;
        BYE from p to s2;
        ...
        do dealWithResult5(p,s1,s2,s3,s4,s5);
    }
}

aux global protocol dealWithResult5( role p, role s1
                , role s2, role s3, role s4, role s5 ) {
    choice {
        RESULT from s1 to p;
        do dealWithResult5(p,s1,s2,s3,s4,s5);
    } or {
        RESULT from s2 to p;
        do dealWithResult5(p,s1,s2,s3,s4,s5);
    } or ...
```

The role p has the choice to send numbers to the other roles; these are the numbers that have to be calculated. The implemented computation logic for role p implements the equal distribution of the numbers that have to be calculated and also the collecting of the perfect numbers. In this example the numbers are equally divided among roles s1 to s5. In this protocol there are three messages, the messages CALC, BYE and RESULT. The CALC message contains a range of numbers that have to be calculated; this is called the group size. The BYE message indicates the sending has stopped, and the RESULT message is sent

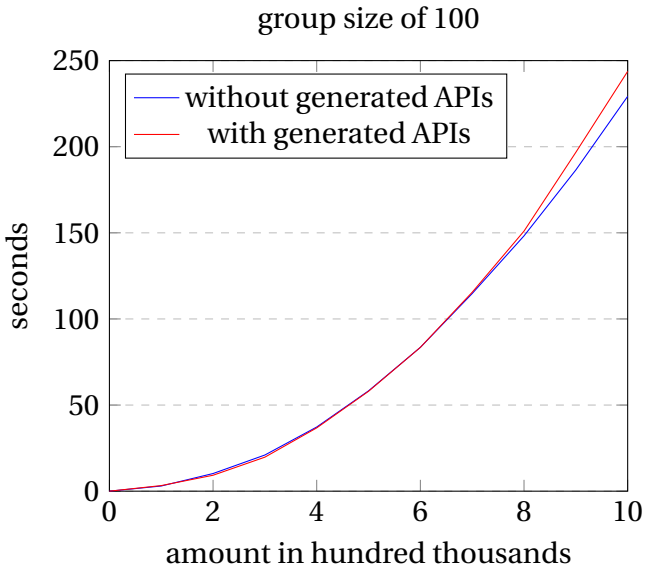

Figure 4.1: Performance graph with amount of numbers divided in groups of 100

back to p with the involved number when this number is perfect.

For the performance testing the group sizes 100, 1000 and 10000 were executed. First 100000 numbers were calculated, then 200000 and this amount increases with steps of one hundred thousand until it reaches one million numbers. Figure 4.1 shows the result of a group size of one hundred. The horizontal axis shows the amount of numbers, for example the number two means two hundred thousand calculated numbers. The vertical axis shows the time in seconds it took to calculate for a given amount of numbers. The red line is the code of the proposed framework and the blue line shows an implementation without the use of generated coordination code. The differences are very small, but with a group size of one hundred the generated code becomes slower when the total amount of numbers increases: about five seconds difference with a total amount of numbers of one million. The tests were taken four times and the average was taken to draw the graph and formulate this result.

The main difference between the different group sizes is the communication between the roles. With a group size of one hundred with a total amount of numbers of one hundred thousand, there will be one thousand CALC messages. With a group size of one thousand there will be one hundred CALC messages.

Figure 4.2 shows the results of the perfectnumber example with a group size of one thousand numbers. Also in this example the times do not differ much, but when the total amount of numbers increase to one million, the generated code seems to become slower relative to the implementation without the generated coordination logic.

Figure 4.3 shows the results of the perfect number example with a group size of 10000. With this group size of 10000 it takes ten CALC messages to calculate one hundred thousand numbers. Also in this example can be seen that the performance of the generated code becomes slower relative to the regular code when more numbers are calculated. But this behavior can only be noticed after ninety message interactions or an amount of 900000 numbers; in other words after the processing of 90 CALC messages. But Figure 4.3 also shows that the generated code performs faster than the regular code for the first ninety

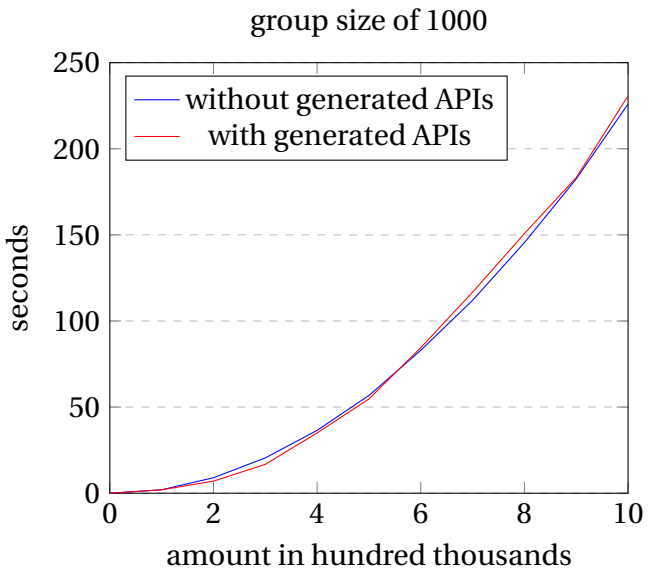

Figure 4.2: Performance graph with amount of numbers divided in groups of 1000

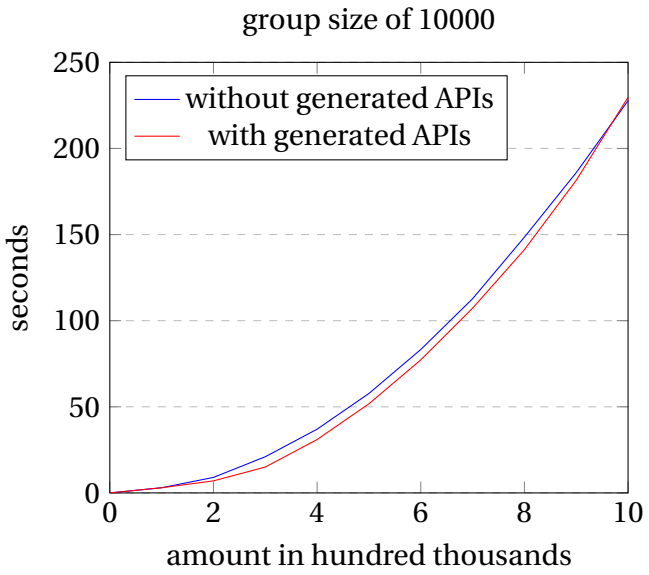

Figure 4.3: Performance graph with amount of numbers divided in groups of 10000

CALC messages. This behavior can only be seen when the total amount of message interactions are below ninety and this situation does not take place with the other group sizes. The message interactions in the 10000 group size are relatively low compared to the other two examples. I have not found an explanation for why the generated code performs a little slower for the first ninety message interactions. This behavior should be more investigated and is future work.

A general conclusion is; when there are more than ninety message interactions the overhead of the generated APIs increases. This can be seen in the Figures 4.1, 4.2 and 4.3. The larger the group size means fewer communications, so the lower the overhead of the generated APIs. The bigger the amount of numbers, the more messages are sent, so the higher the overhead of the generated APIs.

All the measures took place on my laptop that has a one processor with six physical cores and twelve logical cores. As more messages are sent, the code with the generated coordination logic seem to become slower compared to the regular code.

4.6. VALIDATION AND CONCLUSION

Experimental research has been conducted to find the answers to the research questions of this Master thesis. Six concrete example protocols have been tested to validate the proposed solution.

The first example of this project is the Alice and Bob example (1) from Chapter 2.1.3. This example is used in this thesis to illustrate the proposed solution. After this example was implemented with generated coordination logic, other examples followed.

Since the first example was very easy and had only two roles, a role was added to the example. This increased complexity immediately as can be seen in Chapter 3.5.2. The role Fred was added to the example.

The third example used to validate the proposed solution comes from the paper [Yos+13]. In this paper an example based on an online travel agency is discussed. This example has been validated with the proposed solution for TypeScript.

The fourth validated example comes from [Hu17]. This example is called HTTP and is based on the http protocol. There are two roles, client and server and these roles send messages to each other. The messages are based on the HTTP message grammar defined in RFC 7230 [FNR14].

The fifth example comes also from [Hu17] and is called Math Service. This example is implemented with the proposed solution and validated.

The sixth and last example is used to validate the performance of the proposed solution. This example is called perfect number; it is a map reduce pattern. The perfect number example is also discussed in Section 4.5.

These six examples are used to validate the framework for distributed programming in TypeScript based on a global protocol definition.

The conclusion of these validated examples is that the proposed solution works for all of them. For every validated example the framework for programming distributed protocols was generated. The protocol constraints that were defined on the global level were enforced. And facilities for helping the software engineer during the implementation of the computation logic were present. The coordination logic is separated, and the custom computation logic can be passed as parameter to the generated coordination framework, also explained in Chapter 3.

These examples prove that role based distributed programming can be simplified by using a global protocol description. The complete coordination concern of a distributed application can be generated, this way giving a head start during the implementation. Also global level described constraints are enforced and help with implementing the distributed application is achieved with facilities like statement completion.

4.7. PROJECT SCOPE

Some additional remarks regarding the scope of this project should be explicitly made.

The software protocols discussed in this Master thesis are all relatively easy. Protocols with statically known roles can be generated and implemented with the proposed solution. Protocols with dynamically instantiated roles are out of scope. Only global protocols in which message interaction between roles are defined, are within the scope of this master project.

The result of the experimental research conducted for this Master graduation project is discussed in Chapter 3. For this project all implementations were done during this research, except the implementation of a parser for parsing a global protocol definition file. This global protocol definition is layer three which is discussed in Section 3.4. Layer three parses a global protocol definition file and produces a data file with a textual representation of all the involved state machines. This state machine definition file is used by layer two to generate an API for protocol programming. This is discussed in Section 3.3.

4.8. EVALUATION, DISCUSSION AND FUTURE WORK

This project for adding protocol abstractions to TypeScript resulted in a framework that is conceptually modeled in a layered architecture, this is described in Chapter 3. Starting with the simple example called Alice and Bob (1) from Section 2.1.3, other examples followed to validate the framework.

The solution presented in this thesis is based on a DSL from which a statically compiled protocol interface is generated. This protocol API can be used by a software engineer who is responsible for implementing the distributed application.

Coordination code is separated from computation code, which results in higher quality code, as argued for in Section 1.3. Another result is the static validation of protocol constraints in TypeScript and therefore also JavaScript, as explained in Section 2.2. This helps the software engineer who is responsible for implementing the distributed application. And also a headstart is made with building the distributed application. The effort for implementing the coordination logic is reduced to writing a global protocol definition.

The proposed solution could be a powerful technique for developing distributed applications. All the examples that are validated were all server side applications. Applications on mobile devices or Internet browsers were never tried. In theory this should work too, but future work could help confirming this theory.

Also the parsing of the abstraction level three syntax of the global protocol definition to state machines is future work.

Future work could also involve research for directly adding protocol abstractions to the language TypeScript. But also research for adding more complex distributed protocols to current proposed solution for TypeScript is recommended future work.

GLOSSARY

AIOCJ Adaptive Interaction Oriented Choreographies in Jolie.

AOP Aspect Oriented Programming.

API Application programming interface.

BPEL Business Process Execution Language.

DHTML Dynamic HyperText Markup Language.

DOM Document Object Model.

DSL Domain Specific Language.

ECMA European Computer Manufacturers Association.

FSM Finite State Machine.

FTP File Transfer Protocol.

GPL General Purpose Language.

IDE Integrated Development Environment.

IOT Internet Of Things.

JS JavaScript.

JSON JavaScript Object Notation.

MSA Micro Services Architecture.

NPM Node Package Manager.

SOA Service Oriented Architecture.

SoC Separation of Concerns.

STD State-Transition Diagram.

TS TypeScript.

UML Unified Modeling Language.

BIBLIOGRAPHY

ACADEMIC

[Anc+16] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. "Behavioral types in programming languages". In: *Foundations and Trends® in Programming Languages* 3.2-3 (2016), pp. 95–230.

[Arb04] Farhad Arbab. "Reo: a channel-based coordination model for component composition". In: *Mathematical structures in computer science* 14.3 (2004), pp. 329–366.

[Ban+16] Alexey Bandura, Nikita Kurilenko, Manuel Mazzara, Victor Rivera, Larisa Safina, and Alexander Tchitchigin. "Jolie community on the rise". In: *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*. IEEE. 2016, pp. 40–43.

[CG89] Nicholas Carriero and David Gelernter. "Linda in context". In: *Communications of the ACM* 32.4 (1989), pp. 444–458.

[Cia+18] Giovanni Ciatto, Stefano Mariani, Maxime Louvel, Andrea Omicini, and Franco Zambonelli. "Twenty Years of Coordination Technologies: State-of-the-Art and Perspectives". In: *International Conference on Coordination Languages and Models*. Springer. 2018, pp. 51–80.

[Cia96] Paolo Ciancarini. "Coordination models and languages as software integrators". In: *ACM Computing Surveys (CSUR)* 28.2 (1996), pp. 300–302.

[CM13] Marco Carbone and Fabrizio Montesi. "Deadlock-freedom-by-design: multiparty asynchronous global programming". In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 263–274.

[Dal+14] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. "AIOCJ: A choreographic framework for safe adaptive distributed applications". In: *International Conference on Software Language Engineering*. Springer. 2014, pp. 161–170.

[Dij79] Edsger W Dijkstra. "My hopes of computing science (EWD709)". In: *Proceedings of the 4th international conference on Software engineering*. IEEE Press. 1979, pp. 442–448.

[Dra+17a] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: yesterday, today, and tomorrow". In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[Dra+17b] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. "Microservices: How to make your application scale". In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2017, pp. 95–104.

[FNR14] Roy Fielding, Mark Nottingham, and Julian Reschke. "Hypertext transfer protocol (HTTP/1.1): Caching". In: *IETF standardization, RFC 7234* (2014).

[GC92] David Gelernter and Nicholas Carriero. "Coordination languages and their significance". In: *Communications of the ACM* 35.2 (1992), p. 96.

[GDM16] Alberto Giaretta, Nicola Dragoni, and Manuel Mazzara. "Joining Jolie to Docker". In: *International Conference in Software Engineering for Defence Applications*. Springer. 2016, pp. 167–175.

[Gui+17] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. "Microservices: a language-based approach". In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 217–225.

[Haf+16] Munawar Hafiz, Samir Hasan, Zachary King, and Allen Wirfs-Brock. "Growing a language: An empirical study on how (and why) developers use some recently-introduced and/or recently-evolving JavaScript features". In: *Journal of Systems and Software* 121 (2016), pp. 191–208. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.04.045. URL: http://www.sciencedirect.com/science/article/pii/S0164121216300309.

[Hon+11] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. "Scribbling interactions with a formal foundation". In: *International Conference on Distributed Computing and Internet Technology*. Springer. 2011, pp. 55–75.

[Hu17] Raymond Hu. "Distributed Programming Using Java APIs Generated from Session Types". In: *Behavioural Types: from Theory to Tools* (2017), p. 287.

[MGZ14] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. "Service-oriented programming with jolie". In: *Web Services Foundations*. Springer, 2014, pp. 81–107.

[Par94] David Lorge Parnas. "Software aging". In: *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press. 1994, pp. 279–287.

[Pfe16] Rasmus Pfeiffer. "Architecture of Scalable Operating Systems: Multikernel". In: (2016).

[Sil+17] Leonardo Humberto Silva, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, and Anne Etien. "Identifying Classes in Legacy JavaScript Code". In: *Journal of Software: Evolution and Process* 29.8 (2017). e1864 smr.1864, e1864–n/a. ISSN: 2047-7481. DOI: 10.1002/smr.1864. URL: http://dx.doi.org/10.1002/smr.1864.

[Tch+16] Alexander Tchitchigin, Larisa Safina, Manuel Mazzara, Mohamed Elwakil, Fabrizio Montesi, and Victor Rivera. "Refinement types in jolie". In: *arXiv preprint arXiv:1602.06823* (2016).

[Tur59] Herbert W Turnbull. *The Correspondence of Isaac Newton.: 1661-1675*. University Press, 1959.

[Yos+13] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. "The Scribble protocol language". In: *International Symposium on Trustworthy Global Computing*. Springer. 2013, pp. 22–41.

[Zha16] Hongwen Zhang. "Planet of the things". English. In: vol. 2016. 3. 2016, pp. 16–17. ISBN: 1361-3723.

NON-ACADEMIC

[bus18] businessinsider.nl. *the-10-most-popular-programming-languages*. 2018. URL: https://www.businessinsider.nl/the-10-most-popular-programming-languages-according-to-github-2018-10/?international=true&r=US (visited on 11/18/2018).

[cod18] codinghorror. *The Principle of Least Power*. 2018. URL: https://blog.codinghorror.com/the-principle-of-least-power/ (visited on 11/12/2018).

[fre18] freecodecamp.org. *best-programming-languages-to-learn-in-2018*. 2018. URL: https://medium.freecodecamp.org/best-programming-languages-to-learn-in-2018-ultimate-guide-bfc93e615b35 (visited on 11/18/2018).

[hac18] hackernoon.com. *top-3-most-popular-programming-languages-in-2018*. 2018. URL: https://hackernoon.com/top-3-most-popular-programming-languages-in-2018-and-their-annual-salaries-51b4a7354e06 (visited on 11/18/2018).

[Mic12] Microsoft. *TypeScript*. 2012. URL: https://www.typescriptlang.org/index.html (visited on 10/07/2018).

[Net18] Netflix. *Netflix Conductor: A microservices orchestrator*. 2018. URL: https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40 (visited on 10/23/2018).

[npm18] npm. *Node Package Manager*. 2018. URL: https://www.npmjs.com (visited on 10/15/2018).

[out18] outsource.com. *Event Loop in Node.js*. 2018. URL: https://blog.outsource.com/2018/09/26/understanding-the-event-loop-in-node-js-outsource/ (visited on 11/12/2018).

[red18] redmonk.com. *language-rankings-1-18*. 2018. URL: https://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/ (visited on 11/18/2018).

[Scr18] Scribble. *Scribble: Describing Multi Party Protocols*. 2018. URL: http://www.scribble.org/docs/scribble-java.html (visited on 11/12/2018).

[use18] usersnap.com. *programming-languages-2018*. 2018. URL: https://usersnap.com/blog/programming-languages-2018/ (visited on 11/18/2018).

A

CODE EXAMPLES

A.1. MESSAGE SERVER

Listing A.1: Local server for receiving messages(receiveMessageServer.ts).

```
import * as http from 'http';
import {Message} from './Message';
import {messageDB} from './messageDB';

function httpRestServer(req:http.IncomingMessage,res:http.ServerResponse):void {
   const httpHeaders = {'cache-control':'no-cache','Content-Type':'application/json','charset'
   if ( req.method === 'POST' ) {
      let postData:string;
      req.on('data', (data) => { postData = (postData===undefined)?data: postData+data; });
      req.on('end',  () => { try { //console.log(`bericht ontvangen  ${postData}`);
                                   messageDB.add(<Message>(JSON.parse(postData)));
                                   res.writeHead(200, "OK", httpHeaders);
                                   res.end();
                             }
                             catch (err) {
                                   res.writeHead(400, "wrong message", httpHeaders);
                             } } );
      return;
   }
   res.writeHead(404, "page not found", httpHeaders);
}

var httpServer:http.Server = http.createServer(httpRestServer);

function start(port:number){
    httpServer.listen(port);
}

function terminate(){
        setTimeout(
           () => { httpServer.close();
```

```
                    console.log('server is afgebroken, het protocol wordt nu geeindigd');
33                }, 5000 );
}
35
const receiveMessageServer = {
37    start: start
,   terminate:terminate
39 }

41 export {receiveMessageServer};
```

A.2. MESSAGE CLIENT

Listing A.2: Function for sending messages(sendMessage.ts).

```
1  import * as request from 'request';
   import {Message} from './Message';
3  import {connectedRoles,roles} from './globalObjects';

5  export async function sendMessage (roleFrom:roles,roleName:roles,msg:Message):Promise<vo
       //console.log(`send ${msg.name} Message to ${roleName}`);
7      const host = connectedRoles.getInfo(roleName).host;
       const port = connectedRoles.getInfo(roleName).port;
9      //console.log(`host ${host}   port ${port}`);
       msg.from = roleFrom;
11     const options = { url: `http://${host}:${port}`,
                         headers: {'cache-control':'no-cache','Content-Type':'application/j
13                       body: msg,
                         json: true };
15     return new Promise(
           (resolve,reject) => {
17                 request.post( options,
                       (err) => {
19                         if (err) {
                               console.log(`error sendMessage ${roleName} : ${err}`);
21                             reject(err);
                           }
23                         else {
                               resolve();
25                         }
                       }
27                 )
           }
29     );
   }
```

A.3. GLOBAL OBJECTS

Listing A.3: Global needed objects(globalObjects.ts).

```
import {ROLEMESSAGE,Message} from './Message';
import {receiveMessageServer} from './receiveMessageServer';
import {sendMessage} from './sendMessage';
import {messageDB} from './messageDB';

//
//
// verschillende mogelijke rollen
enum roles {
    mediator='Mediator'
}

export {roles}

//
// connected roles
//
type connectedRoleInfo = {
    host: string;
    port: number;
}

const connectedRolesMap:Map<roles,connectedRoleInfo> = new Map();

function showConnectedRoles(){
    connectedRolesMap.forEach(
        (val,key)=>{
           console.log(`${key}   ${val.host}   ${val.port}`);
        }
    );
}

function getInfo(roleName:roles):connectedRoleInfo{
    let roleInfo=connectedRolesMap.get(roleName);
    if (roleInfo) {
        return roleInfo;
    }
    throw new SyntaxError(`${roleName} does not exist`)
}

function missing(nm:roles):boolean{
    return !connectedRolesMap.has(nm);
}

function save(msgOrRoleName:Message|roles,host?:string,port?:number):void{
    switch(typeof msgOrRoleName){
        case 'string':
            if(host&&port) {
```

```
                connectedRolesMap.set(msgOrRoleName, {port:port, host:host});
            }
            break;
        case 'object':
            let roleMsg:ROLEMESSAGE=<ROLEMESSAGE>msgOrRoleName;
            connectedRolesMap.set(roleMsg.roleName, {port:roleMsg.port, host:roleMsg.host
            break;
    }
}

function del(role:roles){
    connectedRolesMap.delete(role);
}

const connectedRoles = {
    missing: missing
,   getInfo:getInfo
,   save: save
,   delete: del
,   showConnectedRoles: showConnectedRoles
}

connectedRolesMap.set(roles.mediator, {port:30000, host:'localhost'});

export {connectedRoles};

async function sleep(ms:number) {
    return new Promise((resolve) => setTimeout(resolve, ms));
}

// generic start function
export async function initialize(roleName:roles, port:number, host:string){
   receiveMessageServer.start(port);
   connectedRoles.save(roleName,host,port);
   await sendMessage( roleName, roles.mediator, new ROLEMESSAGE(roleName) );
   while ( true ) {
       const msg = await messageDB.remove((m)=>m.name===ROLEMESSAGE.name);
       if ( msg.name === ROLEMESSAGE.name) {
           connectedRoles.save(msg);
       };
       const missingRoles = Object.values(roles).filter( (role) => connectedRoles.missing
       if ( missingRoles.length === 0){
           break;
       }
   }
}

//
// custom exception for a runtime check
```

```
export class OneTransitionPossibleException extends Error {
    constructor(public message: string) {
        super(message);
        this.name = 'Exception';
        this.message = message;
        this.stack = (<any>new Error()).stack;
    }
    toString() {
        return this.name + ': ' + this.message;
    }
}

enum roles {
    bob = "Bob",
    alice = "Alice"
}
```

A.4. MESSAGE STORE

Listing A.4: The message database(messageDB.ts).

```
import {Message} from './Message';

interface MessageDB {
    add(msg: Message): void;
    remove(predicate: (msg: Message) => boolean): Promise<Message>;
}

class MessageCoreDB implements MessageDB {
    private items: Message[] = [];
    private predicate: ((item: Message) => boolean) | null = null;
    private resolve: ((item: Message) => void) | null = null;

    add(item: Message): void {
        this.items.push(item);
        this.tryResolve();
    }

    async remove(predicate: (item: Message) => boolean): Promise<Message> {
        let promise = new Promise<Message>((resolve, reject) => {
            this.predicate = predicate;
            this.resolve = resolve;
        });
        this.tryResolve();
        return promise;
    }

    private tryResolve(): void {
        if ( this.predicate && this.resolve ) {
```

```
            let item: Message|null = null;
            let index = 0;
            for (let msg of this.items) {
                if (this.predicate(msg)) {
                    item = msg;
                    break;
                }
                index++;
            }
            if (item) {
                this.resolve(item);
                this.items.splice(index, 1);
                this.predicate = null;
                this.resolve = null;
            }
        }
    }
}

const messageDB:MessageDB = new MessageCoreDB();

export {messageDB}
```

A.5. MESSAGE DEFINITIONS

Listing A.5: Messages from Alice and Bob example(Message.ts).

```
import {connectedRoles,roles} from './globalObjects';

export abstract class Message {
    public from:roles=roles.mediator;
    constructor(public name:string){};
}

export class ROLEMESSAGE extends Message {
    public host:string;
    public port:number;
    constructor(public roleName:roles){
        super(ROLEMESSAGE.name);
        this.host = connectedRoles.getInfo(roleName).host;
        this.port = connectedRoles.getInfo(roleName).port;
    }
}

export class ADD extends Message {
    constructor(public value1: number, public value2: number) {
        super(ADD.name);
    }
}
```

```
export class RES extends Message {
    constructor(public sum: number) {
        super(RES.name);
    }
}
export class BYE extends Message {
    constructor() {
        super(BYE.name);
    }
}
```

A.6. GENERATED API FOR ALICE

Listing A.6: generated API for the role Alice(Alice.ts).

```
import { receiveMessageServer } from "./receiveMessageServer";
import { ADD, BYE, RES, Message } from "./Message";
import { sendMessage } from "./sendMessage";
import { roles, initialize, connectedRoles, OneTransitionPossibleException } from "./globalOb
import { messageDB } from "./messageDB";

enum messages {
    ADD = "ADD",
    BYE = "BYE",
    RES = "RES"
}

interface IAlice {
}

interface IAlice_S1 extends IAlice {
    readonly messageFrom: roles.bob;
    readonly messageType: messages.RES;
    message?: RES;
    send_ADD_to_Bob(add: ADD): Promise<IAlice_S2>;
    send_BYE_to_Bob(bye: BYE): Promise<IAlice_S3>;
}

interface IAlice_S2 extends IAlice {
    recv(): Promise<IAlice_S1>;
}

interface IAlice_S3 extends IAlice {
}

abstract class Alice implements IAlice {
    constructor(protected transitionPossible: boolean = true) { }
    ;
    protected checkOneTransitionPossible() {
```

```
        if (!this.transitionPossible)
            throw new OneTransitionPossibleException("Only one transition possible from a
        this.transitionPossible = false;
    }
}

class Alice_S1 extends Alice implements IAlice_S1 {
    readonly messageFrom = roles.bob;
    readonly messageType = messages.RES;
    constructor(public message?: RES) {
        super();
    }
    async send_ADD_to_Bob(add: ADD): Promise<IAlice_S2> {
        super.checkOneTransitionPossible();
        await sendMessage(roles.alice, roles.bob, add);
        return new Promise(resolve => resolve(new Alice_S2));
    }
    async send_BYE_to_Bob(bye: BYE): Promise<IAlice_S3> {
        super.checkOneTransitionPossible();
        await sendMessage(roles.alice, roles.bob, bye);
        return new Promise(resolve => resolve(new Alice_S3));
    }
}

class Alice_S2 extends Alice implements IAlice_S2 {
    constructor() {
        super();
    }
    async recv(): Promise<IAlice_S1> {
        try {
            super.checkOneTransitionPossible();
        }
        catch (exc) {
            return new Promise((resolve, reject) => reject(exc));
        }
        const msgPredicate: (message: Message) => boolean = m => (m.name === RES.name &&
        const msg = await messageDB.remove(msgPredicate);
        return new Promise(resolve => {
            switch (msg.name + msg.from) {
                case RES.name + roles.bob: {
                    resolve(new Alice_S1((<RES>msg)));
                    break;
                }
            }
        });
    }
}

class Alice_S3 extends Alice implements IAlice_S3 {
```

```
84      constructor() {
            super();
86          receiveMessageServer.terminate();
        }
88  }

90  type Alice_Start = IAlice_S1;
    type Alice_End = IAlice_S3;
92
    async function executeProtocol(f: (Alice_Start: Alice_Start) => Promise<Alice_End>, host: stri
94      console.log(`Alice started ${new Date()}`);
        await initialize(roles.alice, port, host);
96      let done = await f(new Alice_S1());
        return new Promise<Alice_End>(resolve => resolve(done));
98  }

100 export { IAlice, IAlice_S2, messages, Alice_Start, Alice_End, executeProtocol, roles };
```

A.7. GENERATED API FOR BOB

Listing A.7: generated API for the role Bob(Bob.ts).

```
1   import { receiveMessageServer } from "./receiveMessageServer";
    import { ADD, BYE, RES, Message } from "./Message";
3   import { sendMessage } from "./sendMessage";
    import { roles, initialize, connectedRoles, OneTransitionPossibleException } from "./globalObj
5   import { messageDB } from "./messageDB";

7   enum messages {
        ADD = "ADD",
9       BYE = "BYE",
        RES = "RES"
11  }

13  interface IBob {
    }
15
    interface IBob_S1 extends IBob {
17      recv(): Promise<IBob_S2 | IBob_S3>;
    }
19
    interface IBob_S2 extends IBob {
21      readonly messageFrom: roles.alice;
        readonly messageType: messages.ADD;
23      message: ADD;
        send_RES_to_Alice(res: RES): Promise<IBob_S1>;
25  }
```

```
27 interface IBob_S3 extends IBob {
       readonly messageFrom: roles.alice;
29     readonly messageType: messages.BYE;
       message: BYE;
31 }

33 abstract class Bob implements IBob {
       constructor(protected transitionPossible: boolean = true) { }
35     ;
       protected checkOneTransitionPossible() {
37         if (!this.transitionPossible)
               throw new OneTransitionPossibleException("Only one transition possible from a
39         this.transitionPossible = false;
       }
41 }

43 class Bob_S1 extends Bob implements IBob_S1 {
       constructor() {
45         super();
       }
47     async recv(): Promise<IBob_S2 | IBob_S3> {
           try {
49             super.checkOneTransitionPossible();
           }
51         catch (exc) {
               return new Promise((resolve, reject) => reject(exc));
53         }
           const msgPredicate: (message: Message) => boolean = m => (m.name === ADD.name &&
55         const msg = await messageDB.remove(msgPredicate);
           return new Promise(resolve => {
57             switch (msg.name + msg.from) {
                   case ADD.name + roles.alice: {
59                     resolve(new Bob_S2((<ADD>msg)));
                       break;
61                 }
                   case BYE.name + roles.alice: {
63                     resolve(new Bob_S3((<BYE>msg)));
                       break;
65                 }
               }
67         });
       }
69 }

71 class Bob_S2 extends Bob implements IBob_S2 {
       readonly messageFrom = roles.alice;
73     readonly messageType = messages.ADD;
       constructor(public message: ADD) {
75         super();
```

```
    }
    async send_RES_to_Alice(res: RES): Promise<IBob_S1> {
        super.checkOneTransitionPossible();
        await sendMessage(roles.bob, roles.alice, res);
        return new Promise(resolve => resolve(new Bob_S1));
    }
}

class Bob_S3 extends Bob implements IBob_S3 {
    readonly messageFrom = roles.alice;
    readonly messageType = messages.BYE;
    constructor(public message: BYE) {
        super();
        receiveMessageServer.terminate();
    }
}

type Bob_Start = IBob_S1;
type Bob_End = IBob_S3;

async function executeProtocol(f: (Bob_Start: Bob_Start) => Promise<Bob_End>, host: string, po
    console.log(`Bob started ${new Date()}`);
    await initialize(roles.bob, port, host);
    let done = await f(new Bob_S1());
    return new Promise<Bob_End>(resolve => resolve(done));
}

export { IBob, IBob_S2, messages, Bob_Start, Bob_End, executeProtocol, roles };
```

A.8. PERFECT NUMBER GLOBAL PROTOCOL DEFINITION

Listing A.8: Global protocol description of perfect number example(perfectNumberGlobalProto1.scr).

```
module perfectNumber;

global protocol perfectNumber( role p, role s1, role s2, role s3, role s4, rol
    choice {
        CALC from p to s1;
        do perfectNumber(p,s1,s2,s3,s4,s5);
    } or {
        CALC from p to s2;
        do perfectNumber(p,s1,s2,s3,s4,s5);
    } or {
        CALC from p to s3;
        do perfectNumber(p,s1,s2,s3,s4,s5);
    } or {
        CALC from p to s4;
        do perfectNumber(p,s1,s2,s3,s4,s5);
```

```
16      } or {
            CALC from p to s5;
18          do perfectNumber(p,s1,s2,s3,s4,s5);
        } or {
20          BYE from p to s1;
            BYE from p to s2;
22          BYE from p to s3;
            BYE from p to s4;
24          BYE from p to s5;
            do dealWithResult5(p,s1,s2,s3,s4,s5);
26      }
    }
28
    aux global protocol dealWithResult5( role p, role s1, role s2, role s3, ro
30      choice {
            RESULT from s1 to p;
32          do dealWithResult5(p,s1,s2,s3,s4,s5);
        } or {
34          RESULT from s2 to p;
            do dealWithResult5(p,s1,s2,s3,s4,s5);
36      } or {
            RESULT from s3 to p;
38          do dealWithResult5(p,s1,s2,s3,s4,s5);
        } or {
40          RESULT from s4 to p;
            do dealWithResult5(p,s1,s2,s3,s4,s5);
42      } or {
            RESULT from s5 to p;
44          do dealWithResult5(p,s1,s2,s3,s4,s5);
        } or {
46          BYE from s1 to p;
            do dealWithResult4(p,s2,s3,s4,s5);
48      } or {
            BYE from s2 to p;
50          do dealWithResult4(p,s1,s3,s4,s5);
        } or {
52          BYE from s3 to p;
            do dealWithResult4(p,s1,s2,s4,s5);
54      } or {
            BYE from s4 to p;
56          do dealWithResult4(p,s1,s2,s3,s5);
        } or {
58          BYE from s5 to p;
            do dealWithResult4(p,s1,s2,s3,s4);
60      }
    }
62
    aux global protocol dealWithResult4( role p, role s1, role s2, role s3, ro
64      choice {
```

```
        RESULT from s1 to p;
        do dealWithResult4(p,s1,s2,s3,s4);
    } or {
        RESULT from s2 to p;
        do dealWithResult4(p,s1,s2,s3,s4);
    } or {
        RESULT from s3 to p;
        do dealWithResult4(p,s1,s2,s3,s4);
    } or {
        RESULT from s4 to p;
        do dealWithResult4(p,s1,s2,s3,s4);
    } or {
        BYE from s1 to p;
        do dealWithResult3(p,s2,s3,s4);
    } or {
        BYE from s2 to p;
        do dealWithResult3(p,s1,s3,s4);
    } or {
        BYE from s3 to p;
        do dealWithResult3(p,s1,s2,s4);
    } or {
        BYE from s4 to p;
        do dealWithResult3(p,s1,s2,s3);
    }
}

aux global protocol dealWithResult3( role p, role s1, role s2, role s3 ) {
    choice {
        RESULT from s1 to p;
        do dealWithResult3(p,s1,s2,s3);
    } or {
        RESULT from s2 to p;
        do dealWithResult3(p,s1,s2,s3);
    } or {
        RESULT from s3 to p;
        do dealWithResult3(p,s1,s2,s3);
    } or {
        BYE from s1 to p;
        do dealWithResult2(p,s2,s3);
    } or {
        BYE from s2 to p;
        do dealWithResult2(p,s1,s3);
    } or {
        BYE from s3 to p;
        do dealWithResult2(p,s1,s2);
    }
}

aux global protocol dealWithResult2( role p, role s1, role s2 ) {
```

```
114     choice {
            RESULT from s1 to p;
116         do dealWithResult2(p,s1,s2);
        } or {
118         RESULT from s2 to p;
            do dealWithResult2(p,s1,s2);
120     } or {
            BYE from s1 to p;
122         do dealWithResult1(p,s2);
        } or {
124         BYE from s2 to p;
            do dealWithResult1(p,s1);
126     }
    }
128
    aux global protocol dealWithResult1( role p, role s1 ) {
130     choice {
            RESULT from s1 to p;
132         do dealWithResult1(p,s1);
        } or {
134         BYE from s1 to p;
        }
136 }
```