Imperial College London

MENG INDIVIDUAL PROJECT FINAL REPORT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Web Development Based On Multiparty Session Types in Purescript

Author:
Hei Yin Fong

Supervisor:
Nobuko Yoshida

Second Marker:
Iain Phillips

June 15, 2020

Submitted in partial fulfillment of the requirements for the Type of degree of Imperial College London

Abstract

Web technology advancements bring evolutionary changes to the user experience of web applications. New technology like WebSockets simplifies the communication between clients and servers, leading to the emergence of increasingly complex web applications, which involve huge amount of data exchanging over the communication channel. This creates new challenges for developers to ensure that the implementation of their applications conforms to the complex communication protocol.

In this project, we introduced a novel web development work flow which allows developers to give types to a communication protocol, namely *Session Types*, and utilise the types as a static guidance to build a communication error-free application in Purescript. The target of our work is extending the framework to improve its usability and capability, for example, improving the error reporting as the program fails to follow the protocol, and merging WebRTC, a powerful real-time communication interface, into the framework. We also conducted a set of benchmarks to assess the performance of the framework and investigated about a compile time overhead issue which prevented the framework to be used to an industry standard.

Acknowledgments

I would like to thank my supervisor, Nobuko Yoshida, for giving me the opportunity to work on this project, and I am incredibly grateful for her guidance throughout the course of the project.

I would also like to thank Francisco Ferreira Ruiz and Fangyi Zhou for their support, and particularly, Fangi Zhou for taking his time to provide feedback for my final report.

Lastly, I need to thank Jonathan King for giving me the resources to learn Purescript and answering my questions about the Session Runtime Library.

Contents

Chapter 1

Introduction

The development of web applications has moved from serving up static data from a database to data that is created on-the-fly following an event trigger, creating a real-time user experience. This transformation needs new web solutions, as the traditional way of retrieving static web contents using *stateless* HTTP communication protocols are no longer sufficient to support the higher standard of responsiveness and greater workload, because of the latency caused by cycles of establishing one-request-one-response connections. Insight of this, *stateful* web applications that allow full-duplex persistent connections are developed, improving performance by reducing the number of connections per request and enabling the *states* of a user's session to be stored and retrieved on the web server. The advantages of *stateful* web applications range from creating dynamic web pages to improving users' experiences, thanks to the new technologies such as Web-Sockets[18] and WebRTC[19], which are communication channels that support browser-server connections and browser-browser connections respectively.

Stateful web applications could provide desktop-applications like experiences to users from within browsers, however, this also indicates interactions between end-points for example browser to server could become very complex where communication errors like deadlocks could occur easily, leading to an unexpected error or incorrect behaviour. To avoid this, a pre-agreed communication protocol, which describes the rules and format of digital messages, could be set up between two ends to ensure the correctness of the communication. However, there were no formal models of communication protocols to reason about the conversation behaviour, until the birth of *session types*.

Session types arise to give types to the communication patterns stated in a protocol and verify the protocol through tools like *Scribble*. The emergence of *session types* is originated from studies of concurrent computing, where researchers study how concurrent processes should communicate with each other. Seeing this as an effective tool to put forward the development of interactive web applications, many have been exploring ways to put this theory into practice through implementing libraries that support *session types* in mainstream languages such as Java, Python, Purescript and so on.

A recent paper contributed by Jonathan King, Nicholas Ng and Nobuko Yoshida proposes a novel approach to embed *session types* in Purescript [14] by leveraging its strong type system. Purescript is a functional language that compiles to Javascript, which is used for building web applications, server side applications and desktop application with the use of Electron [28]. The paper presents a workflow that uses *Scribble* to generate code based on *Multiparty Session Types (MPST)* theory and a lightweight *session runtime* library for the actual implementation of session types.

Built on [14], this project explores the various possibilities that focus on the practical aspect of the work flow presented in the paper, ranging from implementing new features to improve the user experience to integrating new web technologies to expand the usage of the lightweight session runtime library introduced by the work.

1.1 Contributions Outline

In this project, we improved the usability and functionality of the Session Runtime Library for building multiparty session type-safe web applications in Purescript. The specific contributions are stated in the below.

- **Improving Error Messages**: Type errors originating from protocol violations with the use of session combinators in the Session Runtime Library are obscure, and lack clear information as to what causes the error. We implemented an extension to produce better error messages by extending the code generation of Scribble-Purescript to generate code that is able to report customised errors when a protocol infringement happens.
- **Direct Client-to-Client Communication via WebRTC**: We implemented a WebRTC option in the Session Runtime Library for users to construct browser-to-browser sessions, and to send and receive arbitrary data over the data channel of a WebRTC session.
- **Benchmark of Purescript SessionRuntime Library**: We investigated the compilation performance of the library through implementing numerous benchmarks, targetting different features of a protocol. We also looked into a significant compile time overhead issue arising from extensive branching.

1.2 Structure of Report

- **Chapter 2 - Background**: Contains information needed to understand *Session Types* and *Scribble Code Generation*.
- **Chapter 3 - Purescript Code Generation**: Introduces the code generation process of the work presented in [14] to build multiparty session type-safe web applications in Purescript. This part provides the required information to understand the rest of the report.

- **Chapter 4 - Improving Error Messages**: Details the problem with the original type errors from protocol violation, and the implementation of the solution to produce type errors that convey useful information for debugging.
- **Chapter 5 - WebRTC Client-to-Client Session**: Consists of introduction of WebRTC and implementation details of extending the Session Runtime Library to allow a WebRTC session to be constructed.
- **Chapter 6 - Benchmark of Purescript Session Runtime Library**: Explains the content and results of a set of benchmarks completed for evaluating the Session Runtime Library and investigating a compile time overhead issue from branching.
- **Chapter 7 - Evaluation**: Assesses the compilation performance of the error reporting feature and the quality of the WebRTC feature.
- **Chapter 8 - Conclusion**: Summarises the project and proposes future directions.

1.3 Github Repositories

- **Improving Error Messages**: `https://github.com/haileyfong/scribble-java/tree/hyf/purescript-codegen`
- **WebRTC Client-to-Client Session**: `https://github.com/haileyfong/purescript-scribble/tree/hailey`

Chapter 2

Background

2.1 Session Types

2.1.1 Overview

In this world of distributed and concurrent computations, communication between processes or systems is nevertheless the core component in supporting their operations. With the rising need for more complex systems, system engineers find it increasingly difficult to ensure the absence of communication errors such as deadlocks, mismatch of message type. To address these problems, there is a need for a new form of the communication design framework.

In the past decades, there are many studies done on communication techniques, and the two main abstractions are shared memory and message passing, of which message passing can be modeled by process π-calculus proposed by Robin Milner [7], which describes that processes communicate with each other via exchange of messages.

Session types [8] are types that structure communication sequences. It is a notion originated from processes theory based on asynchronous communication which based on π-calculus. It is modeled by primitives in π-calculus for sending, receiving and continuation of sessions. The initial session types only supported binary sessions, where only two participants are involved in the communication session, this work was further extended to multiparty sessions [9], supporting a wider range of practical communication-centred applications.

With session types, a structured protocol agreed among communicating parties can be established and checked for deadlocks where two processes waiting for responses from each other to make progress and communication mismatch where one process receives an unexpected type of messages from other processes. Programs performing communications can also be guaranteed to conform to protocols and are free from the errors mentioned previously.

The use of session types facilitates the development of correct and efficient communication-

centric programs with more reliable communication sessions. Session types have been implemented in a vast variety of mainstream programming languages, including Java[2], Python[10], Purescript, C[11], Go and etc. Other interesting applications are motion session types for robotics interaction[12].

2.1.2 Multiparty Session Types

Before multiparty session types were introduced, binary sessions, as the first session types proposed, were used to compose many communication patterns, where only two participants are described in a session. Figure 2.1 shows an example binary session, two processes in the session are associated with `Customer` and `Agency` respectively, which represent a scenario of a customer trying to place an order through an agency. The communication between them is modeled as a sequence of synchronised message exchanges, which is as follows.

1. The customer sends an order to the agency, for example, *Iceland*, a place where the customer would like to travel to.

2. Upon receiving the order from the customer, the agency looks up on his system to find a quotation for the desired destination, *Iceland*, and sends the result to the customer.

3. Once the quote is received at the customer, there are two options the customer can go with: (1) accept the quoted price if the customer thinks the price is reasonable (2) rejects the quote if it is overly priced.

4. If the customer decides to accept it, he sends an acceptance message to the agency as a confirmation, makes a purchase and terminates the protocol. If the customer refuses to go ahead, he sends a rejection message to the agency and terminates the session.

Binary session types, however, have limited power to express all communications which involve more than two parties and are insufficient to validate the correct interactions. Hence, in order to extend the expressive power of binary session types, multiparty session types come into play, providing a better expressiveness through *global type* and *local type*.

In a nutshell, multiparty session types [9][13] is a type theory that verifies more than two concurrent message passing processes. To verify these multiparty interactions, two new methodology *global type* and *local type* are introduced.

- **Global type** describes the global communication behaviour of all involved parties from a bird-eye's perspective, enabling checking of protocol conformance.

- **Local type** describes the communication behaviour of a specific participant from its perspective.

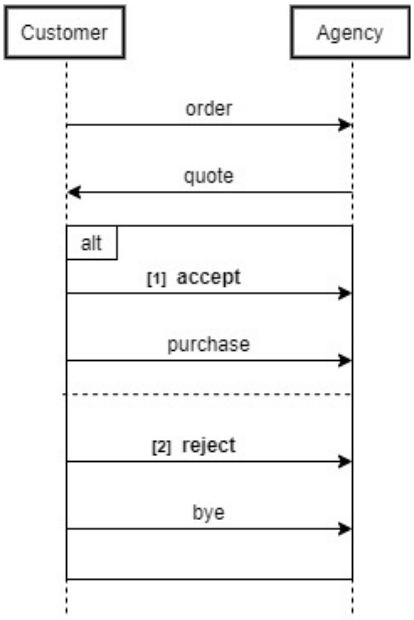


Figure 2.1: A graphical representation of an example binary session

Global type, also known as global view, can be projected to each end-point participant's view, obtaining a local type, which by composing them together creates a set of message *send* and *receive* that are permitted by the global type. These projected end-point protocols can then be used to check that implementations of each participant comply with the global protocol, which is guaranteed free from communication safety errors, such as unexpected message receptions and deadlocks.

Consider another example depicted in Figure 2.2 involving three processes characterised by `Customer`, `Agency` and `Hotel Manager`, will be expected to obtain the following pattern of message exchanges, which will then be translated into *global type*.

$$\begin{aligned}
&\text{Customer} \rightarrow \text{Agency} : \mathit{order}(\mathit{String}).\\
&\text{Agency} \rightarrow \text{HotelManager} : \mathit{quote}(\mathit{Int}).\\
&\text{Agency} \rightarrow \text{Customer} : \mathit{quote}(\mathit{Int}).\\
&\text{Customer} \rightarrow \text{Agency} : \{\\
&\mathit{accept}().\text{Agency} \rightarrow \text{HotelManager} : \mathit{accept}().\text{Customer} \rightarrow \text{Agency} : \mathit{purchase}(\mathit{Int}),\\
&\mathit{reject}().\text{Agency} \rightarrow \text{HotelManager} : \mathit{reject}().\text{Customer} \rightarrow \text{Agency} : \mathit{bye}(\mathit{Int})\\
&\}
\end{aligned}$$

The type stated in the above describes the global pattern of message passing between all participants in the communication session, which comprised of sequencing of message *send*, *receive* and *choice*. The basic pattern `Customer` → `Agency`*:m(S)* indicates a message *send* named *m* from `Customer` to `Agency` which carries data of type *S*. The communication starts by `Customer` sending an order to `Agency`, then,

`Agency` sends a quote to both `HotelManager` and `Customer`. After receiving a quote from `Agency`, `Customer` needs to decide whether or not he should accept it. Therefore, there are two possible continuations, either accept it or reject it . In the case of accept, `Customer` sends an *accept* message to `Agency`, and `Agency` does the same to `HotelManager`. Finally, `Customer` completes the session by sending a *purchase* message with the purchase amount. Otherwise, `Customer` sends a *reject* message to `Agency`, `Agency` also sends a *reject* message to `HotelManager`, and the session terminates by the *bye* message sent by `Customer`. The . operator and the , operator denote sequencing of communication patterns and the different possible continuations respectively.

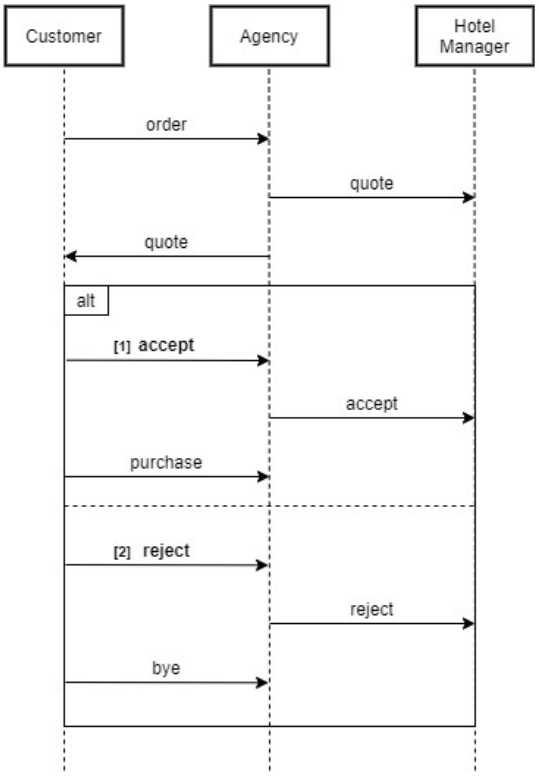


Figure 2.2: A graphical representation of an example global protocol of multiparty session types

2.1.3 Scribble and Endpoint API generation

Overview

Scribble[1] is a protocol specification language and a code generation framework, which applies multiparty session types theory to practice. It provides guidelines for the application of session type principles and techniques to modern software development. Scribble is used to describe how different entities interact with each other in a communication formally. In a protocol laid out in Scribble, every communicating party is known as *Role* which exchanges typed messages with *name* and *signature*. *Name* labels the message so that different messages can be distinguished without reading the data carried by them, whereas *signature* gives the type of the payload data in the message.

Example Global Protocol Specification

The below illustrates an example global protocol specification, conceptually very similar to the *global type* mentioned in section 2.1.2, for a restaurant in the Scribble language, where we explain the syntax and structure of Scribble protocols. In the example, the first line declares the module of the protocol with the keyword `module` followed by the module name `Restaurant`. As Scribble organises protocols into modules, this tells which module does the protocol belong to. What follows after are declarations of message payload types. Scribble is designed to work with data types from other languages, therefore, in line 3 and 4, external data types `IntType` and `StringType` from Python are imported and renamed as `Int` and `String`, so that they could be used later for specifying the message payload type in the protocol. Given that this is a specification of a global protocol, in line 6, the keyword `global` and `protocol` are required, followed by a protocol signature `Serve`.

This protocol involves two roles, `Waiter` and `Customer`, aliased as `W` and `C` respectively, which together establish a session. At the beginning, `Waiter` sends a `Greeting` message of the type `String` to `Customer`. Then, `Customer` responds by asking for the menu via a `AskForMenu` message. Here, there is not a payload type specified which indicates an empty message. `Waiter` sends the menu through a `Menu` message with the payload type `String Array`. After receiving a list of available food options on the menu, `Customer` will decide whether an order should be placed. This is expressed by the syntax `choice at`, which corresponds to the branching behaviour of the following interaction. There are two possible actions from `Customer`, either placing a food order by sending a `Order` message of type `String` or terminating the session by sending a `Bye()` to `Waiter`.

Protocol Verification and Projection

After the global protocol is specified, the Scribble toolchain checks the well-formedness of the protocol. If no errors are thrown, the protocol is well-formed and will be pro-

```
1  module Restaurant
2
3  type <py> "types.IntType" from "types.py" as Int;
4  type <py> "types.StringType" from "types.py" as String;
5
6  global protocol Serve(role Waiter as W, role Customer as C)
7  {
8      Greeting(String) from W to C;
9      AskForMenu() from C to W;
10     Menu([String]) from W to C;
11     choice at C {
12         Order(String) from C to W;
13     } or {
14         Bye() from C to W;
15     }
16 }
```

Figure 2.3: Global Serve Protocol

jected into separate local protocols for each role, as shown in Figure 2.5. A local protocol is a *local type* mentioned in Section 2.1.2, which is a localised version of the input global protocol from the view of a particular participant, thus, it will contain only interactions that involve the target role. A Scribble local protocol at `Customer` resulted from the projection of the global protocol in Figure 2.3 is shown in Figure 2.5. We can see that only interactions that involve `Customer` appear in the protocol. All the payload types in the global protocol are preserved, and the protocol is decorated with the keyword `local` instead of `global` in line 6. An `at` keyword followed by the protocol name`Serve_Customer` indicates from which viewpoint this local protocol is. A functionality provided by Scribble is automatically generating code from the projected protocols independent of other local protocols as an implementation of each role, which is free from communication errors given by the typed structure. Scribble could also represent these endpoint protocols as Endpoint Finite State Machines (EFSMs), which contain *states* and *transitions*, representing the states in the protocol and interactions between roles respectively. EFSMs could act as guidelines [14] for software developers to implement the communication aspect of their applications.

Scribble API Generation

As mentioned in the above, Scribble could perform code generation from end-point protocols, a result of the projection of the global protocol. What code generation actually does is generating APIs that offer functionalities for the target language to conduct conversations that conform to the local protocols, this conformance is only possible if the language's type system is powerful enough to make this guarantee through static type check. It is believed that if every end-point program that re-

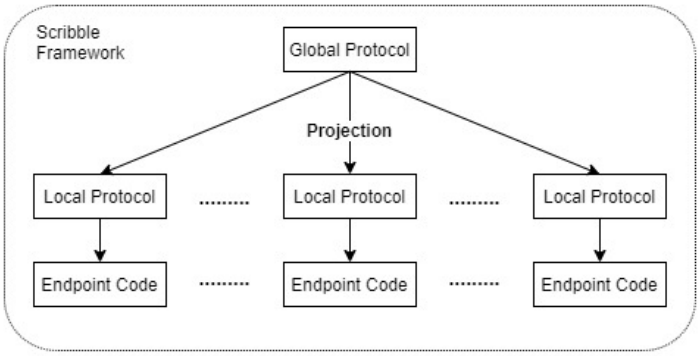


Figure 2.4: The projection of global protocol specification by the Scribble Framework

```
module Restaurant_Customer

type <py> "types.IntType" from "types.py" as Int;
type <py> "types.StringType" from "types.py" as String;

local protocol Serve_Customer at Customer
(role Waiter as W, role Customer as C)
{
    Greeting(String) from W;
    AskForMenu() to W;
    Menu([String]) from W;
    choice at C {
        Order(String)to W;
    } or {
        Bye()to W;
    }
}
```

Figure 2.5: Local Protocol

spects its corresponding local protocol, the entire program obeys the global protocol and the overall correctness is guaranteed based on multiparty session type theory. Hence, the use of the generated APIs gives communication safety to the program theoretically. With that said, only programming languages with a strong enough type system are able to achieve this without extra assistance, however, many mainstream languages such as Python lack this property such as alias restrictions that ensure linear usage of communication channels, which is an essential element of implementing session types. As a result, *dynamic verification* [15], also known as run-time check, is introduced to validate the conversation behaviour of the program at run-time.

The code generation process differs depending on the kind of programming language. Taking Java as an example, a Finite State Machine(FSM) encoding the control flow of a protocol is created for each end-point protocol, an example of this is

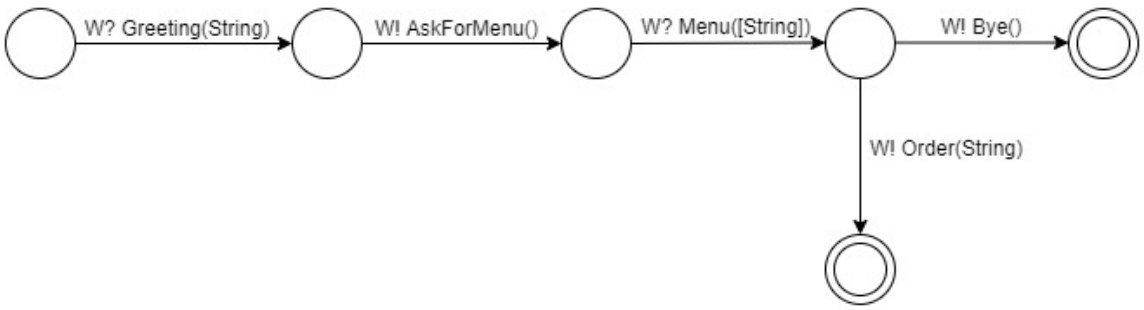


Figure 2.6: The FSA generated from the local protocol by Scribble

shown in Figure 2.6, then each state will be converted into a class, and all outgoing transitions will be treated as its methods. The methods will contain communication primitives to send and receive messages, and return the next state in the state machine. This way the developer can use a chain of methods starting from the initial state to reach a terminating state without intermediate states. When all roles implement the generated protocols correctly, the interacting roles will have the guarantees from the theory of multiparty session types, that communication errors including session fidelity and deadlocks become impossible combined with run-time checks. More code generation methods are described in Section 2.2.2.

2.2 Session Types Implementation

Session types theory has been broadly studied in the past decades, with a vast variety of different implementations targeting different programming languages. These implementations could be split into two categories, one focusing on functional languages that does not use code generated from Scribble, another one uses Scribble directly as a code generation tool. Both of them are detailed in the following sections.

2.2.1 Implementation of Session Types in Functional Languages with linearity

Embedding session types naively in programming languages has been a prolonged challenge. There are different challenges arising from different types of programming languages in order to ensure linearity, the linear usage of channel resources, which must be guaranteed during the implementation of session types. Functional languages are well-known for their strong typing discipline, which is very instrumental in upholding the linearity property in the session types' implementation. There are many different techniques developed and are listed in the following.

EFSM encoding using type classes in Purescript

Purescript is a functional language that compiles to Javascript, which is commonly used in web applications development. A framework introduced in [14] presents a novel way of implementing session types in Pure-script with static linearity, meaning non-linear usage of channels is prevented at compilation. One ground-breaking technique is the encoding of End-point Finite State Machine(EFSM) generated from Scribble using multi-parameter type classes and the session-runtime library derived from the encoding that ensures linear channel usage. The summary of the proposed workflow for developing type-safe web applications is as follows.

1. The whole process begins with using Scribble, where a global protocol is specified and verified for its well-formedness.
2. Then, end-point protocols and their corresponding EFSMs are produced from the projection of the well-formed global protocol, of which EFSMs will be encoded into data types using multi-parameter type classes: states and transitions are interpreted as types and type-class instances. Type classes are used to enable overloading of functions, where a type is polymorphic but constrained to be an instance of the type class. Common examples are `Eq a`, `Order a` and `Monad a`. The use of multi-parameter type classes instead of single-parameter type classes is to encode relations between states using *functional dependencies*, where states are represented as the type parameters in the definition of the multi-parameter class.

3. Deriving from the EFSMs, new data types and type-classes are created to represent states and transitions in the EFSMs. These types act as a static guidance for users to use session runtime to program the communication behaviors in the application that conform to the specified global protocol. The session runtime is a library of communication combinators that can only be used to construct correct protocol implementations. They are connect/disconnect, send/receive, choice/branch and session.

This work is one of the pioneering work that introduces the multiparty session-typed code generation workflow targeting interactive web applications with a static linearity guarantee. This innovative technique presented is original and does not inherit from any previous work.

Model-View-Update-Communicate Session Types in Link

With the growing complexity of web applications, new architecture patterns like Elm architecture and the Model-View-Update (MVU) application pattern have gained more popularity than traditional Model-View-Controller design pattern. More specifically, the MVU architecture inspired by Elm architecture, targets web applications with interactive GUI and gives a much simpler and straightforward architecture combined with functional programming.

Simon Fowler in this paper [16] proposes the first formal model of the MVU architecture as a concurrent π-calculus, and uses it as a foundation for embedding session types that supports the implementation of a MVU architecture using Link, a functional language that is designed to facilitate web programming by providing a single language for all tiers(web browser, web server, back-end services). It tackles the difficulty of enforcing linear usage of channels with applications integrated a GUI by introducing two extension, *linearity*, to implement session types safely, and *model transitions*, to allow multiple message types. *Linearity* in particular is implemented through defining an *extraction* function which returns a linear model whenever there is an update to the model.

Session-Ocaml

Session-ocaml [17] implements session types in OCaml utilising OCaml specific data structures *lenses* and *slot* `monads`, where *slots* are nested tuples and *lenses* are combinators that provide access to a particular element in *slots*. It provides static linearity, and multiple supports simultaneous sessions but is limited to binary session types. Session-type inference for checking if a protocol type matches its counterpart is enabled via unification in OCaml with *polarised session types*.

2.2.2 Scribble-based code generation

Code generation from Scribble is one of the effective ways of implementing session types in programming languages. It is a technique that generates APIs directly from the projected end-point protocols, as demonstrated in Figure 2.7, which could then be used in writing end-point programs with all the properties guaranteed by the multiparty session types theory. There are many different approaches developed specific to different languages which are listed in the below.

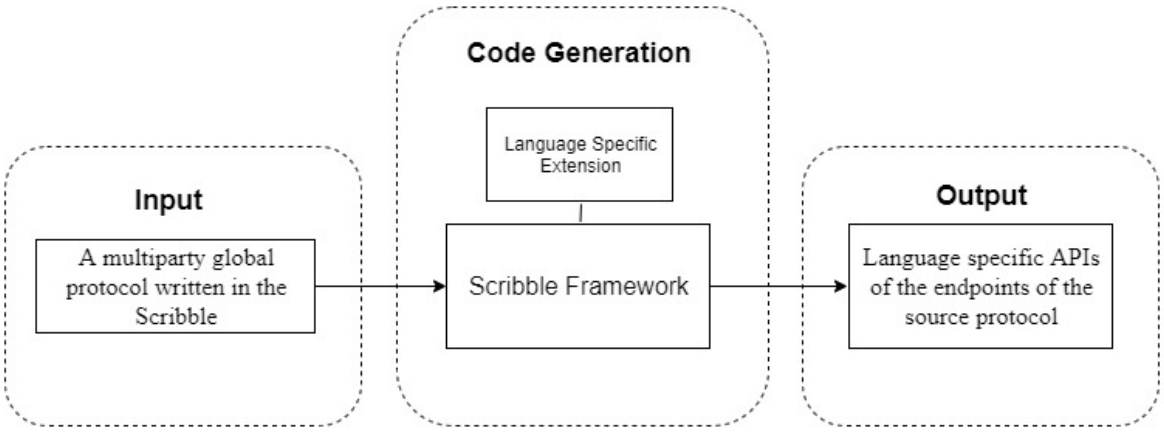


Figure 2.7: An abstraction of the Scribble-based code generation process

Scribble-Java

Scribble-Java [2] is the first piece of work that presents *hybrid session verification* for applying session types theory directly to programming languages. It is an extension to the initial Scribble framework with an API generation technique that statically verifies the correct communication behaviour represented in the input protocol, this is done by exploiting the native type system of the target language, Java. Apart from the static type checking, run-time checks are also used to ensure linear usage of channels, which garbage-collects any used channel resources by regularly examining the flag of the resources. The proposed approach targets desktop applications and only supports those using TCP or HTTP as communication protocols.

The Endpoint API generation of each role in the protocol is summarised in the following.

- Endpoint API is generated from a Finite State Machine (FSM) of the I/O actions to be performed in the protocol by the target role.
- An Endpoint API comprises a family of Java classes that represent the states of the protocol relevant to the target role, with methods for the I/O actions permitted in each state.

Scribble-Scala

Scribble-Scala [3] is again an extended Scribble framework that targets the implementation of full multiparty session types(projection, type-merging, and delegation) in Scala, a language that combines object-oriented and functional programming. It uses Scribble to generate Scala APIs by decomposing multiparty session types. The generated APIs use the type-safe binary channels provided by the `lchannel` library to perform communications that enforce typing of I/O actions through the Scala type system. However, similar to Scribble-Java, the capabilities of the static typing in the language are insufficient in guaranteeing linear usage of channel resources, thus, run-time checks are also needed for the enforcement of linearity. The implementation uses Akka actor framework and supports transport abstractions like TCP, local memory and Akka actors.

Pabble

Pabble [5] is a protocol language and a tool that generates deadlock-free *Message Passing Interface* (MPI) parallel backbone code from Parameterised Scribble. The proposed code generation framework begins with specifying a global protocol in Pabble, the same as Scribble-Java and Scribble-Go. Then, an MPI source code is automatically generated from the global protocol definition and is merged with the sequential code describing the application behaviour to produce a final MPI program, using LARA, an aspect-oriented programming (AOP) language. LARA can also occasionally optimise generated MPI source code by overlapping communication patterns to reduce unnecessary duplications. Instead of verifying the correctness of the generated code via type-checking, conformance of the specified global protocol is ensured upon auto code generation, which also ensures code to be error-free.

Scribble-Go

Extending from Scribble, Scribble-Go [6] role-parametric session types. The Scribble-Go tool-chain guarantees statically well-typed endpoint The approach mentioned in Pabble was revisited using a new distributed formalisation of parameterised Scribble and applied to Go in the Scribble-Go toolchain, with support for TCP and in-process shared memory transport. Generates APIs based on CFSMs.

Session-type Provider in F#

[4] features a *session-type provider* library developed in F# that performs on-demand compile-time protocol validation and code generation of protocol-specific .NET types for users' end-point implementations, extending from Scribble. Its implementation of session types in F# gives a static linearity through type checking of generated types for messages and I/O operations. *Session-type provider* is integrated with Scribble which generates end-point specific APIs from the global protocol specification

with refinements. The generated interface provides methods that chain I/O actions to reach a target state. The following is briefly how the code generation operates.

- Each protocol state is generated as a distinct .NET class type, with methods that give exactly the permitted I/O actions at that state.
- Each I/O method's return type corresponds to the successor state of that action.

Chapter 3

Purescript Code Generation

This project is focused on extending the work on creating multiparty session type-safe web applications in Purescript by Jonathan King[14]. Therefore, it is important to know the details of King's work in order to understand the rest of the report.

King introduced a work flow that begins with specifying a global protocol in Scribble, then using the generated types from Scribble to program an endpoint implementation with a Session Runtime library, which allows protocol violations to be detected during compilation. The implementation involves presenting a new type-level encoding of the Endpoint Finite State Machine(EFSM) using multi-parameter type classes, extending Scribble to generate Purescript source code that is based on the encoding, and building a Session Runtime library which consists of session combinators for developers to program The Session Runtime library leverages the Purescript type system to type check protocol conformance. Figure 3.1 shows an overview of the entire process.

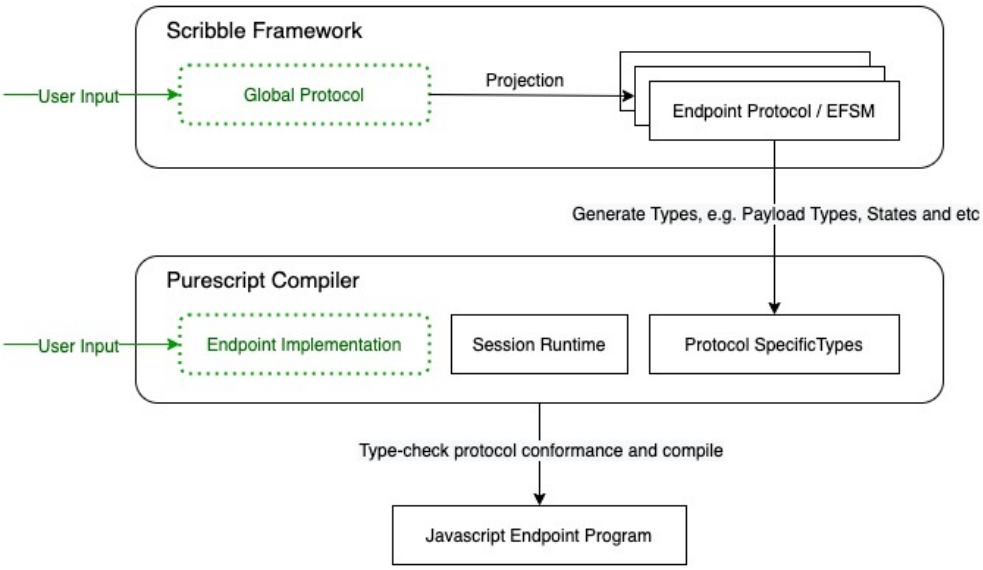


Figure 3.1: Overview of the Development Work Flow

3.1 Global Protocol Specification

Imagine you are developing a simple web application that involves two clients, A and B, and a server, where the server relays every message sent by client A to client B. The first step is to write a global protocol in Scribble that represents the overall communication structure of every party in the application, as shown in Figure 3.2. This way you could make sure communication errors, for instance deadlocks, are absent in your application, if the input protocol passes the check in Scribble, which examines the well-formedness of global protocols.

```
1  module Relay
2
3  type <purescript> "String" from "Prim" as
       String;
4
5  global protocol Relay(role ClientA, role
       ClientB, role Server)
6  {
7      Connect() connect ClientA to Server;
8      Connect() connect ClientB to Server;
9      choice at Server {
10         Message(String) from ClientA to
               Server;
11         Message(String) from Server to
               ClientB;
12     } or {
13         Quit() from ClientA to Server;
14     } or {
15         Quit() from ClientB to Server;
16     }
17 }
```

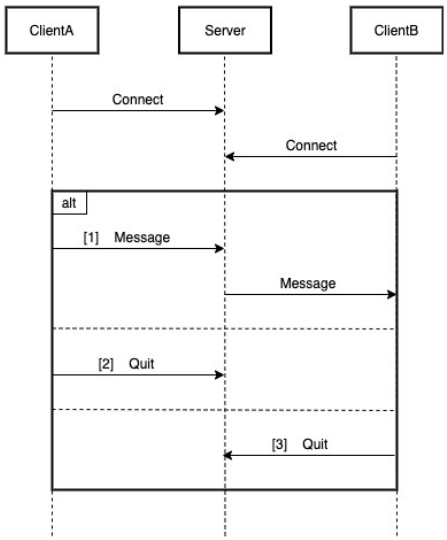


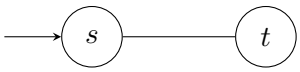
Figure 3.2: Example Scribble Global Protocol(Left), Graphical Representation of the Protocol(Right)

3.2 Generate Types via EFSM Encoding

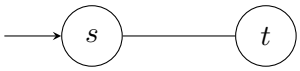
3.2.1 Encoding EFSM Transitions as Multi-parameter Type Classes

To allow programs to be type-checked against the projected endpoint protocols, Endpoint Finite State Machines from Scribble is integrated into the endpoint program by representing *states* and *transitions* in EFSMs as *types* and *instances* of multi-parameter type classes in Purescript. There are different type classes for different kinds of transitions, which will be discussed in the following.

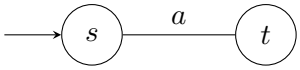
```
class Connect (r :: Role) (r' :: Role) s t | s ⇝ r r' t
```



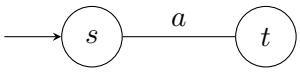

```
class Disconnect (r :: Role) (r' :: Role) s t | s ⇝ r r' t
```



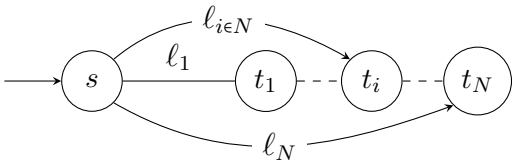

```
class Send r s t a | s ⇝ t r a
```




```
class Receive r s t a | s ⇝ t r a
```




```
class Select r s! (ts :: RowList) |s! ⇝ ts r
```




```
class Branch r  r s  (ts :: RowList) |s ⇝ ts r  r
```

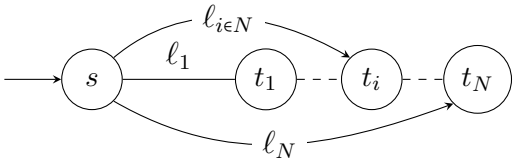


Figure 3.3: Type Class Definitions of EFSM Transitions

The use of type classes is to provide type constraints for its corresponding functions in the Session Runtime library, so that the use of those functions is restricted to follow the communication pattern from the EFSM. To illustrate how are transitions encoded and how could they impose the protocol conformance in the session runtime, three kinds of type classes are displayed in Figure 3.3: (1) `Connect` and `Disconnect` for establishing and terminating connections, (2) `Send` and `Receive` for transmitting data between communicating parties, and (3) `Select` and `Branch` for label selections and branching.

Encoding Send and Receive

The second row and third row show the type class definitions of `Send` and `Receive`, which are the interpretations of a *send* and *receive* transition respectively. Both type classes consist of type parameters *s*, *t* and *a* which represents the current state, the successor state and the message payload type of the transition individually. Functional Dependencies are used in the type class declarations, such that some type parameters uniquely determine another. In the definitions of `Send` and `Receive`, the current state *s* decides the successor state *t*, the role of the recipient *r* and the mes-

sage payload type *a*, with the annotation $s \rightsquigarrow t\ r\ a$. This encodes the relationship between the different properties of a transition, and enforces that exactly one successor state, one type of message and one particular role are involved in a `Send/Receive` transition given a current state.

Encoding Connect and Disconnect

The transitions of connecting and disconnecting from an entity are encoded as `Connect` and `Disconnect`, as depicted in the first and second row in Figure 3.3. Different from `Send` and `Receive`, there exists a type parameter `r'` which indicates the role of the party being connected/disconnected to from role of type *r*. Similar to `Send` and `Receive`, both type classes consist of the initial state *s* and successor state *t* type parameters, where the initial state *s* determines the succesor state *t* and which two roles *r* and *r'* are being connected/disconnected with each other via functional dependencies $s \rightsquigarrow r\ r'\ t$.

Encoding Select and Branch

In the case of label selection and branching, there could be multiple transitions possible from a given current state, and each transition is distinguished by a message payload label. As shown in the third and fourth row in Figure 3.3, we see that given a current state *s* could lead to numerous continuation states t_i, which are wrapped inisde the type parameter *ts*, a `RowList` containing tuples of a type-level string ℓ_i and a continuation state t_i . Exactly one transition is going to be picked and this depends on the message payload label ℓ_i. All these selection and branching actions are usually followed by another transition, for instance *send*, which is determined by the successor state of the previous *select/branch* transition, and inductively determined by the message payload label. This creates a selection and branching of next possible actions through sending a label, then performing the corresponding action. Similar to the above, type parameters in the type class declarations are annotated with functional dependencies, where current state *s* indicates the list of possible continuation states *ts* and the role receiving the label *r* in `Select` and the roles sending and receiving the label `r` and `r'` in `Branch`.

3.2.2 Types and Instances Generation from Scribble

Having input a global protocol to the Scribble Framework in the first step, Purescript code is generated from Scribble based on the encoding of EFSM mentioned previously. The generated code contains new data type for every state and message payload, and type class instance for every transition. The types and instances generated form a **static guidance** for developers to build a web application that follows the input protocol using the **Session Runtime combinators**, which have the corre-

sponding type constraints. Based on the global protocol described in Figure 3.2, the respective generated code is shown in Figure 3.4 and Figure 3.5.

```
data Message = Message String
derive instance genericMessage :: Generic
   Message _
instance encodeJsonMessage :: EncodeJson
   Message where
  encodeJson = genericEncodeJsonWith
     jsonEncoding
instance decodeJsonMessage :: DecodeJson
   Message where
  decodeJson = genericDecodeJsonWith
     jsonEncoding

data Quit = Quit
derive instance genericQuit :: Generic Quit _
instance encodeJsonQuit :: EncodeJson Quit
   where
  encodeJson = genericEncodeJsonWith
     jsonEncoding
instance decodeJsonQuit :: DecodeJson Quit
   where
  decodeJson = genericDecodeJsonWith
     jsonEncoding
```

Figure 3.4: Sample Generated Payload Types

Message Types Generation

Every message payload in the input protocol results as a new data type in the generated code. Referring to Figure 3.4, the generated data types for the message payload `Message` and `Quit` in the protocol are shown, which are the substitues of the message payload type parameter *a* during an instance initiation of the above type classes `Send` and `Receive`. During the transmission of messages, data contained in every message has to be in a Json format, therefore, both `Message` and `Quit` have instances instantiated for `EncodeJson` and `DecodeJson`, which hold functions for encoding and decoding Json objects, known as `encodeJson` and `decodeJson` respectively.

State Types and Transition Instances Generation

As mentioned previously, a new data type is created for each state in the EFSM, and for every transition, an instance corresponding to the kind of the transition is generated. In Figure 3.5, instances `initialClient` abd `terminalClient` represent the transitions for initialising and terminating a session. Instance `connectS11` represent the transition for connecting `ClientA` to the `Server`. Instance `selectS13` indicates a `select` action performed to the `Server`, given that the successor state of `select` depends on the message label, we see a mapping of string label to state types in `("message" :: S13Message , "quit" :: S13Quit)`, where `"message"` corresponds to the successor state `S13Message`. Figure 3.6 shows the EFSM of the input protocol with transitions annotated with the respective generated instances.

```
foreign import data S11 :: Type
foreign import data S12 :: Type
foreign import data S13 :: Type
foreign import data S13Message :: Type
foreign import data S13Quit :: Type
foreign import data S14 :: Type

instance initialClient :: Initial ClientA S11
instance terminalClient :: Terminal ClientA S12
instance connectS11 :: Connect ClientA Server S11 S13
instance sendS13Send :: Send Server S13Message S13 Message
instance sendS13Quit :: Send Server S13Quit S14 Quit
instance selectS13 ::
  Select Server S13 ("message" :: S13Message, "quit" :: S13Quit)
instance disconnectS14 :: Disconnect ClientA Server S14 S12
```

Figure 3.5: Sample Generated Types

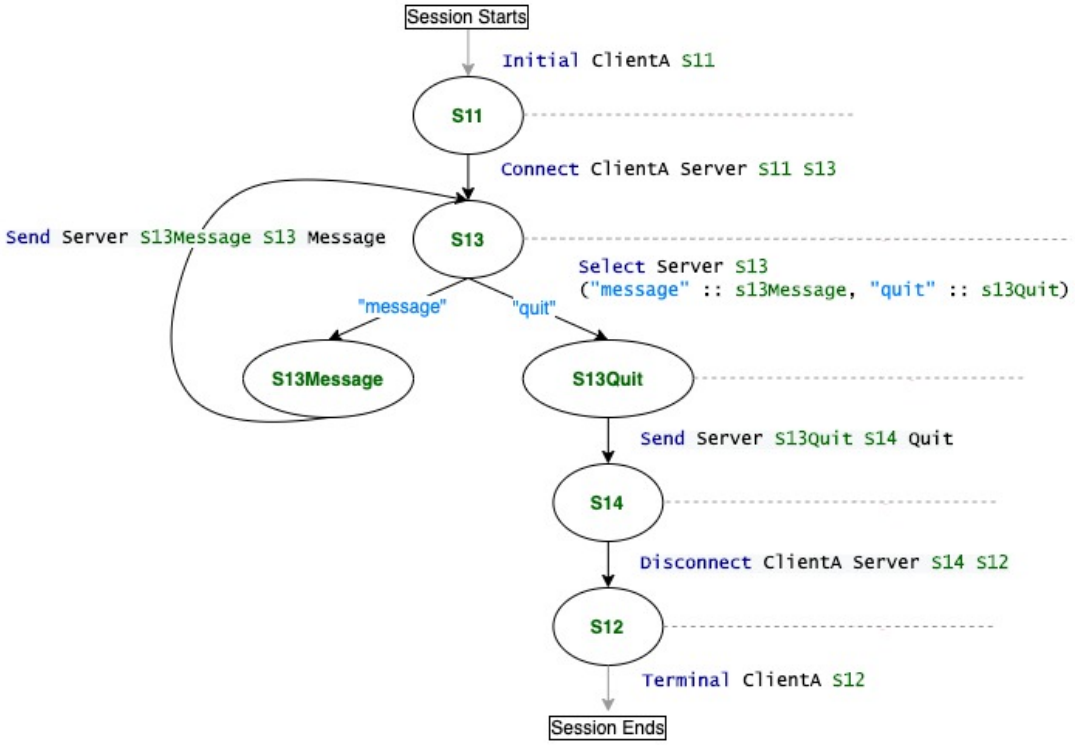


Figure 3.6: Generated Types and its corressponding EFSM

3.3 Endpoint Programming with Session Runtime

Once the code generation from Scribble is completed, the final step is to implement the endpoint using the generated types and instances using the **session runtime combinators**. Every communication action in the input protocol is confined within a **session runtime** in the endpoint program, and the session runtime combinators are there to construct session runtimes that conform to the protocol, including `session`, `request/connect/disconnect`, `send/receive` and `select/choice`, which are shown in Figure 3.7.

3.3.1 Session Runtime Combinators

Looking at the definitions of `choice`, `send` and `receive`, a type constraint responding to the action of the function is applied to each of them: `Branch r r' s ts` for `choice`, `Send r s t a` for `send` and `Receive r s t a` for `receive`, this tells the Purescript compiler to refer to the previously generated types and instances and look for a unique instance(a valid transition) that could satisfy all the constraints applied to the function. If there is not a matching instance, a protocol violation is indicated.

```
session :: forall r c p s t
   m a.
   Transport c
=> Initial r s
=> Terminal r t
=> MonadAff m
=> Proxy c
-> Role r
-> Session m c
-> ma
```

```
choice :: forall r r' rn c s ts u
   funcs row m p a.
   Branch r r' s ts
=> RoleName r' rn
=> IsSymbol rn
=> Terminal r u
=> Transport c p
=> Continuations (Session m c)
   ts u a funcs
=> ListToRow funcs row
=> MonadAff m
=> Record row -> Session m c s u a
```

```
send :: forall r rn c a s t
   m p.
   Send r s t a
=> RoleName r rn
=> IsSymbol rn
=> Transport c p
=> EncodeJson a
=> MonadAff m
=> a -> Session m c s t Unit
```

```
receive :: forall r rn c a s
    t m p.
   Receive r s t a
=> RoleName r rn
=> IsSymbol rn
=> Transport c p
=> DecodeJson a
=> MonadAff m
=> Session m c s t a
```

Figure 3.7: Primitives and Types in the Session Runtime Library

3.3.2 Transport Abstraction

Communications could be performed over `WebSockets` or `Memory` provided by the library. To give the flexibility of adding other transport methods, for example, WebRTC, an API for direct browser to browser connection, the transport layer is generalised by introducing a `Transport` type class, which contains primitives for conducting communication actions like `send`, `receive` and etc. We see that in `session`, `send`, `receive` and `choice`, they all have a constrained type variable *c*, which allows parameterisation of the transport layer of a session runtime. More details of this abstraction is discussed later.

Chapter 4

Improving Error Messages

The following lays out the road map of this chapter.

- Section 4.1 - Background
- Section 4.2 - Problem with the default type error messages
- Section 4.3 - Implementation details of improving error messages with custom errors

4.1 Background

Given an input protocol to Scribble, the generated code serves as a *static* requirement for users of the Session Runtime library to construct sessions that satisfy the protocol. If there is a line of Purescript code violating the input protocol, for example, a *send "hello"* is performed instead of a *receive* which is required by the protocol, a Purescript compilation error will be thrown since a corresponding type class *instance* representing `send "hello"` in not found in the *static* requirement. An example of the error is shown in Figure 4.1.

```
No type class instance was found for

  Scribble.FSM.Send t4
                    S13
                    S13
                    Connect
```

Figure 4.1: An Example **no type instance is found** Error

There are in general two reasons that could cause such errors.

- **Incorrect action**: Performing an unexpected action which is against the input protocol. Using the example in Figure 4.2, the protocol expects a `send` action

followed by a `receive` action, yet, in the user's program in Figure 4.3 , a `send` action is followed by another `send` action, this produces a `no type instance is found` error during compilation.

```
foreign import data S11 :: Type
foreign import data S12 :: Type
foreign import data S13 :: Type

instance sendS12 :: Send Server S11 S12 Ping
instance receiveS14 :: Receive Server S12 S13 Pong
```

Figure 4.2: Fragment of an Example Generated Code of a Ping-Pong Protocol

```
ping = do
...
  send Ping
  send Ping
...
```

Figure 4.3: Fragment of an Example Endpoint Implementation of the Ping-Pong Protocol

- **Incorrect message payload**: Sending or receiving a message payload that is not expected according to the protocol. For example, `send Connect` is required by the protocol once a connection is requested. However, the user mistakenly performed `send "Hello"` instead. This again causes the compiler to throw a `no type instance is found`, because of the mismatch of payload type `Connect` and `String`.

4.2 Problem

Throwing a `no instance is found` error whenever something is breaking the protocol does not do library users good. If the protocol gets more complex, it is undoubtedly causing a lot of pain for users to debug the issue, as the error does not give anymore information than just "a certain instance could not be found". Hence, we think that improving the error reporting of the library could enhance its usability by providing useful information to the users when such error occurs.

4.3 Improving Error Messages with Custom Errors

Our solution to improve error messages is by creating custom type errors supported by Purescript, and inserting them into a bunch of extra instances generated to repre-

sent any actions that are opposed to the protocol, forcing the compiler to throw the custom errors when a protocol violation takes place.

4.3.1 Custom Type Errors

To prompt a different type error messages, we need a way to customise type error messages. Here, we utilise the `Prim.TypeError` module embedded in Pure-script compiler, which is used to produce custom type errors. The module contains a `Fail` type class, which can be used as a type constraint to embed custom type error messages. The following is an example.

```
instance incorrectSend :: Fail(Text "send is not an expected
   action") => Send Server S13 S13 String
```

Figure 4.4: An Example Usage of the `Fail` Type Constraint

If the instance `incorrectSend` is chosen by the compiler during type class constraint solving, the custom error message `"send is not an expected action"` introduced by the `Fail` type constraint will be output by the compiler, as shown by Figure 4.5.

```
A custom type error occurred while solving type class constraints:

  send is not an expected action
```

Figure 4.5: An Example Custom Type Error

As a result, we could make use of this `Fail` type constraint to customise messages to provide more information to the users.

4.3.2 Capture of All Possible Incorrect *Transitions*

To make use of the `Fail` type constraint, we extended the code generation of Scribble-Purescript to generate instances that represent all possible incorrect *transitions* of the EFSM derived from the input protocol, and attach the `Fail` type constraint to these instances to prompt the compiler to output custom type errors when any of these instances is resolved, indicating an occurrence of protocol violation.

First Approach: Incorrect Successor States

The first approach taken is to define *incorrect instances* as instances that have *incorrect* successor state. Therefore, for each *correct* instance, we keep track of a set of states in the EFSM excluding the successor state of the *correct* instance, and use that to generate a set of *incorrect* instances that are differed only by their successor

states. This approach does not work, as it has led to an *overlapping instances* error, an error that occurs when there are more than one suitable instances to be chosen by the compiler, because the constraint solver regards instances of the same type class with the same current state as identical instances, which is defined by the functional dependencies in the type classes' declarations of the Session Runtime Library. An example is shown in Figure 4.6, where instances `sendS12` and `sendS14` are regarded as *overlapping instances*.

```
foreign import data S11 :: Type
foreign import data S12 :: Type
foreign import data S14 :: Type

instance sendS12 :: Send Server S11 S12 String
instance sendS14 :: Send Server S11 S14 String
```

Figure 4.6: An Example of Overlapping Instances

Second Approach: Incorrect Type Classes

The failure from the first approach shows that we need to re-define what an *incorrect* instance means. In the second approach, we defined an *incorrect* instance as an instance that must represent a different action, but has the same current and successor state. The following Figure 4.7 shows an example of all other possible incorrect instances(*transitions*) with the new definition denoted by red arrows with respect to a correct instance(*transition*) `Send Server S11 S12 Message`.

We extended the code generation of Scribble-Purescript to generate the *incorrect* instances. We defined a `CustomErrorInstanceFactory` class with a class method `generate`, which takes an `instance` and returns the corresponding *incorrect* instances. These instances are applied with `Fail` constraints embedded with corresponding error messages, which are parameterised and will be discussed later in section 4.3.3.

During code generation, we traverse the *transitions* of EFSM for every *role*, and create instances with the correct type class and current state, and also recipient role, successor state and message payload type if applicable. Then, for each of these instances, we generate the corresponding *incorrect* instances, by following the below method.

1. Initialise a list with all kinds of transitions.
2. Remove the type of the transition that is being traversed from the list. The remainder of the list represents the types of the *incorrect transitions* with regard to the current *transition*.
3. Iterate over the list and perform the following.

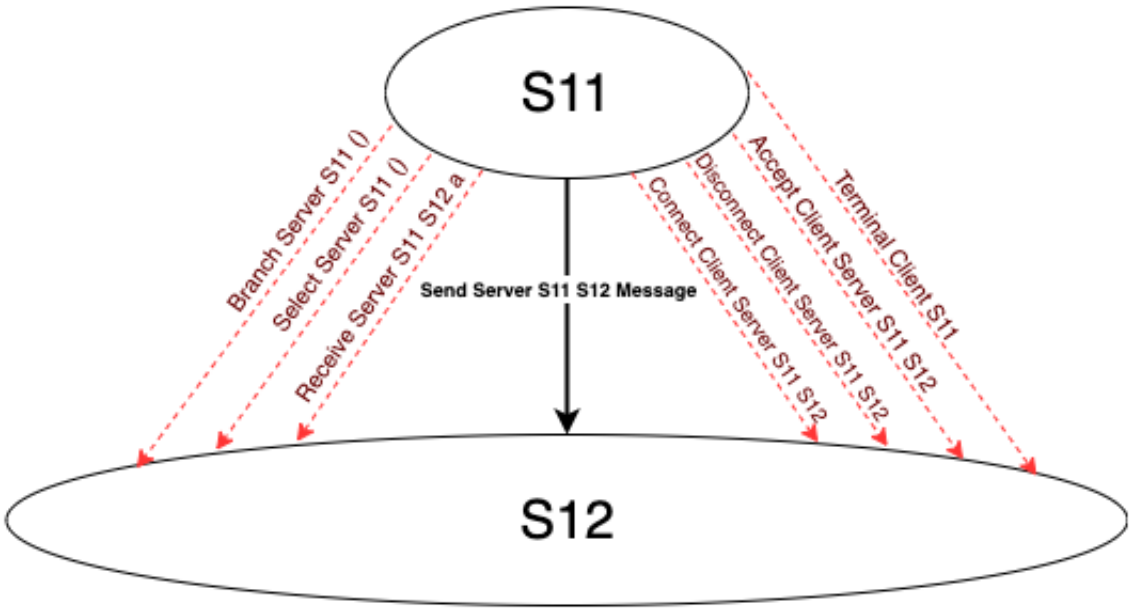


Figure 4.7: An Example of All Possible Transitions. Red dotted arrows indicate incorrect transitions. Black solid arrow represents the expected transition

(a) Generate a custom error message associated with the type of the *incorrect* transition.

(b) Create the type constraints `Fail` which holds the custom error for *incorrect* transitions of type `Send` and `Receive` for providing information about the message payload (This will be discussed further in the next section)

(c) Create an instance for the *incorrect transition* and append it to the output which holds every instance generated throughout the process.

Figure 4.8 shows an example fragment of code generated from a *correct* insrance `Send Server S11 S12 Connect`.

Mismatch of Message Payload Types

The above code generation however does not create an instance for a *correct* action but with an *incorrect* message type, because only instances belonging to a different type class are generated. However, if we try to generate instances with a different payload type but of the same type class and same current state, an `overlapping instances` error is going to be thrown. To mitigate this issue, we applied *instance alternation*[1], a technique to prevent *overlapping stances* by inserting an `else` keyword before `instance`, such that the compiler only considers the else instances when the instances before them are not suitable. An example is displayed in Figure 4.9.

[1]https://github.com/purescript/documentation/blob/master/language/Type-Classes.md

```
instance sendS12 :: Send Server S11 S12 Connect

/*Incorrect instances*/
instance receiveS12Fail :: (Fail(Text " .. "))
  => Receive Server S11 S11 a

instance connectS12Fail :: (Fail(Text " .. "))
  => Connect Client Server S11 S11

instance disconnectS12Fail :: (Fail(Text " .. "))
  => Disconnect Client Server S11 S11

instance acceptS12Fail :: (Fail(Text " .. "))
  => Accept Client Server S11 S11

instance selectS12Fail :: (Fail(Text " .. "))
  => Select Server S11 ()

instance branchS12Fail :: (Fail(Text " .. "))
  => Branch Client Server S11 ()

instance terminalS12Fail :: (Fail(Text " .. "))
  => Terminal Client S11
```

Figure 4.8: Example Fragment of the Generated Code

```
/* correct instance */
instance correctSend :: Send Server S13 S13 Connect

/* incorrect instance */
else instance incorrectSend :: (Fail(Text "Mismatch of payload type" )) =>
    Send Server S13 S13 String
```

Figure 4.9: An Example of Instance Alternation

4.3.3 Structure of Error Messages

The content of the custom error message of each *incorrect instance* is decided and formed during code generation. Each of these custom error messages carries two pieces of information: **the performed action** that led to the error and **the expected action**. Figure 4.10 displays a general structure of the custom error messages. For `Send` and `Receive` specifically, an additional information about the type of message payload is also printed, denoted by a square bracket in the figure.

```
Actual: <Type of Action> [<Type of Message Payload>]
Expected: <Type of Action> [<Type of Message Payload>]
```

Figure 4.10: Structure of an Custom Error Message

Parameterisation of Message Payload Types

From Figure 4.10, we need to extract the message payload type of the action that leads to an error for `Send` and `Receive` instances. This is done by making the message payload type a type variable, and passing the variable to the `Quote` type constructor. The `Quote` type constructor renders any type as a `Doc`, which could be used in the custom type error, as shown in Figure 4.11. Other type constructors seen in the figure, for instance, `Above` and `Beside`, deal with the positioning of the messages.

Figure 4.12 demonstrates an example custom error caused by a mismatch of message payload types.

```
else instance failSend ::
   (Fail(Above (Beside (Text "Actual: Send message of type ") (Quote b)) (
      Text "Expected: Send message of type Add"))))
   => Send Server S13 S13 b
```

Figure 4.11: An Example *Incorrect* Instance with Parameterised Message Payload Type

```
A custom type error occurred while solving type class constraints:

  Actual: Send message of type String
  Expected: Send message of type Add
```

Figure 4.12: An Example *Incorrect* Instance with Parameterised Message Payload Type

Chapter 5

WebRTC Client-to-Client Session

The following lays out the road map of this chapter.

- Section 5.1 - Background
- Section 5.2 - Overview of WebRTC, the Real-time Communication API
- Section 5.3 - Approach of Writing Purescript Bindings for the Javascript WebRTC API
- Section 5.4 - Implementation Details of WebRTC Transport Functions in the Purescript Session Runtime Library
- Section 5.5 - An Example WebRTC Application using the Session Runtime Library

5.1 Background

The Purescript Session Runtime Library uses WebSockets and Memory as the means to transfer data between entities, where Websockets[18] is a technology designed to provide a bi-directional communication channel between a server and a browser. These options refrain from users to construct direct client-to-client sessions using the library, meaning that users are unable to implement any protocols that involve client-to-client interactions. Hence, we see a opportunity to enhance the capabilities of the library by incorporating WebRTC[19] to enable users to create a direct Client-to-Client session.

5.2 Overview of WebRTC

WebRTC is an open-source framework that enables Real-Time communications in browsers. It is embedded in many modern browsers including Google Chrome, Firefox, Safari and Opera, and is also available on native clients for all major platforms

like iOS and Android. It supports direct peer-to-peer connections for sending video, voice and generic data, allowing developers to build high-quality Real-Time applications. WebRTC is implemented as a set of JavaScript APIs, the major components are as follows [19].

- `getUserMedia` gives the permission for acquiring audio and video media input from a device's microphone and camera.
- `RTCPeerConnection` represents the WebRTC connection between peers. It comes with methods to connect to a remote peer, maintain and monitor the connection, and close the connection once it's no longer needed.
- `RTCDataChannel` uses the same API as WebSockets with very low latency, enabling bidirectional communication of arbitrary data between peers.

5.2.1 Creating a WebRTC Connection

Figure 5.1: The flow of establishing a WebRTC connection

In the simplest case, forming a WebRTC connection requires peers(WebRTC clients) and a signalling server. At first, peers do not know about each other, and do not have the needed network information to communicate with each other directly. Therefore, a signalling server is in place for them to exchange the necessary data(including network details) to establish the connection. Once a connection is created, the server is no longer needed by the peers. To illustrate how is the process like, Figure 5.1 shows a simple example of peer Alice attempting to connect to peer Bob via a signalling server.

Exchange of media information: Session Description Protocol

A webRTC connection is viewed as a multi-media session, where video, audio or arbitrary data could be sent/received over the channel. Therefore peers need to agree upon details such as list of media formats to be transmitted, bandwidth information, transport addresses and etc, and this is achieved via a **SDP offer/answer negotation**. As shown in Figure 5.1, Alice as the connection initiator sends an ”offer” in the form of Session Description Protocol (SDP), a standardised representation of the description of a multi-media session, which is widely used in Session Initiation Protocol, Real-time Transport Protocol, and Real-time Streaming Protocol[22]. In the context of WebRTC, it is used to pass all the multimedia-specific metadata, for example, the kind and format of media being transferred, the transfer protocol being used, the IP address and the port of the endpoint and other information needed to describe a multi-media session. This ”offer” is relayed to Bob by the signaling server. Then, once Bob has received the ”offer” from Alice, it responds to Alice by sending an SDP answer, which describes the session of the its side via the signalling server. If both peers have reached an agreement of the description of the session, each peer must hold two descriptions at its end: the **local description** describing itself and the **remote description** describing the other peer. The following Figure 5.2 depicts how the negotiation is carried out using the WebRTC API.

```
Alice = new RTCPeerConnection()     | signalling  | Bob = = new RTCPeerConnection()
offer = createOffer()               |---offer---->| Bob.setRemoteDescription(offer)
Alice.setLocalDescription(offer)    |             |
Alice.setRemoteDescription(answer) |<---answer---| answer = createAnswer()
                                    |             | Bob.setLocalDescription(answer)
```

Figure 5.2: The Use of WebRTC API during the Offer/Answer Negotiation

Exchange of network information: ICE Candidates

Apart from exchanging information about the media, information about the network connection is also required, which is known as ICE candidates. They contain the details of the methods available to the peers to communicate. This process starts by each peer proposing its best candidates and getting down to the worse candidates. Typical candidates are UDP, which are preferred over TCP because they are faster, and recovery from interruptions are easier for media streams. Once these candidates are gathered, only the optimal ICE candidate will be chosen. Then, all of the required metadata, network routing (IP address and port), and media information used to communicate for each peer is agreed upon, and the network socket session between the peers is then fully established and active. Next, local data streams and data channel endpoints are created by each peer, and multimedia data is finally transmitted both ways using whatever bidirectional communication technology is employed.

If the negotiation process of determining the best ICE candidate fails, which does happen sometimes due to firewalls and NAT technologies in use, the fallback is to use a TURN server as a relay instead. This process basically employs a server that acts as an intermediary, and which relays any transmitted data between peers.

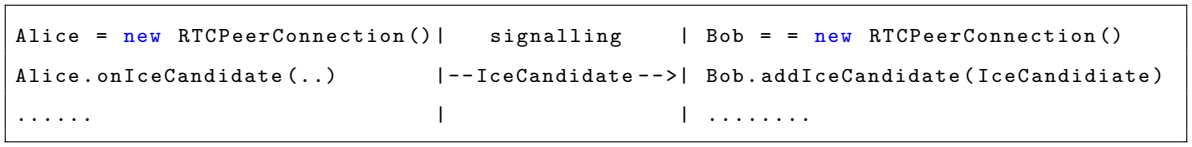


Figure 5.3: The Use of WebRTC API during ICE Candidates Exchange

5.3 Purescript FFI Bindings

Initiating the WebRTC connection and performing communication actions over the connection require calling to the **WebRTC JavaScript API**. To make it accessible for our Purescript library, the first step is to create **Purescript bindings** by using the **Foreign Function Interface (FFI)**, which allows you to call JavaScript functions from Purescript. Writing Purescript FFI bindings is basically giving type to an existing Javascript value using a *foreign import* declaration. That means we need to create a *foreign Javascript module*, whom values are going to be imported to the Purescript module via the *foreign import* declaration. The following demonstrates how this is done.

5.3.1 RTCPeerConnection API

The `RTCPeerConnection` interface provides a wrapper for the connection between the local peer and the remote peer, and holds methods, for example, `createOffer`, `createAnswer`, `setLocalDescription`, `setRemoteDescription` and etc. , for the local peer to establish a connection with the remote peer, and to monitor and close the connection whenever it is needed.

```
connectionConfig = ...
localPeer = new RTCPeerConnection(connectionConfig)
```

Figure 5.4: Example Usage of the `RTCPeerConnection` API

We see that `RTCPeerConnection()` is a constructor that returns a newly created `RTCPeerConnection`. `connectionConfig` is a dictionary of parameters that configures the connection upon connection creation. These parameters are optional, and therefore will not be discussed here.

To call this `RTCPeerConnection` constructor in Purescript, we need to understand how to export Javascript functions to Purescript via the FFI interface. Javascript functions and values are exported from foreign Javascript modules by assigning them to the exports object. In the left below, a `newRTCPeerConnection` Javascripts function is defined and assigned to `exports`. `newRTCPeerConnection` returns a function with no arguments which returns the call to the `RTCPeerConnection` constructor which is wrapped inside. The purpose of this is to delay the evaluation of `new RTCPeerConnection()`, which is only available during runtime, until `newRTCPeerConnection` is called to prevent an *undefined* error. This is mapped to an `Effectful`[1] computation in Purescript, where we assign the type `RTCConfiguration -> Effect RTCPeerConnection` to `newPeerConnection` in Purescript, shown by the following foreign import declaration on the right. Now the Purescript binding for this API is completed. Both modules have to be stored at the same directory with

[1] https://github.com/purescript/purescript-effect

the same prefix, for example, `RTC.js` and `RTC.purs`.

```
/*RTC.js*/
exports.newRTCPeerConnection =
  function(psConfig) {
    return function() {
        return new
           RTCPeerConnection(..);
    };
};
```

```
/*RTC.purs*/
foreign import data
   RTCPeerConnection ::
   Type
foreign import
   newRTCPeerConnection
  :: RTCConfiguration ->
      Effect
     RTCPeerConnection
```

Figure 5.5: Javascript Foreign Module(Left), Foreign Declarations in Purescript(Right)

5.3.2 creatOffer and createAnswer API

`createOffer` and `createAnswer` are methods of the `RTCPeerConnection` interface. Referring to Figure 5.1, these two methods are called during the exchange of media information as a WebRTC connection is being instantiated. The purpose of each function is detailed in the below.

- `createOffer()`: Initiates a *Session Description Protocol* (SDP) offer when starting a new WebRTC connection to a remote peer. The SDP offer contains information about the media data that is going to be transmitted over the connection, and transport parameters needed for the multi-media stream to be constructed [20].
- `createAnswer()`: Creates an SDP answer in response to an offer from a remote peer during the offer/answer negotiation phase of establishing a WebRTC connection. The SDP answer again contains information similar to an SDP offer, but it is a description of the remote end [21].

```
offerPromise = AliceConnection.createOffer([options]);
answerPromise = BobConnection.createAnswer([options]);
```

Figure 5.6: Example Usage of `createOffer` and `createAnswer`

Both methods return a `Promise` object, which if an offer/answer is created successfully, is resolved with a `RTCSessionDescription` object holding the newly created offer/answer. There are optional parameters that could be passed to both functions for customising the offer/answer.

Given that both `createOffer` and `createAnswer` are asynchronous functions, returning a `Promise` object, we model this in the Javascript foreign module by returning

an asynchronous function with error and success callbacks, which are used to handle situations when an offer/answer has failed to create and is created successfully respectively. Similar to `RTCPeerConnection`, `_createOffer` and `_createAnswer` are declared and are assigned to exports objects.

`_createOffer` and `_createAnswer` are defined as functions that take a `RTCPeerConnection` object as parameter. Inside them, `createOffer` and `createAnswer` are called respectively upon `pc`, the `RTCPeerConnection` that is passed to the top-level function. If an offer/answer creation is successful, a `RTCSessionDescription` object is returned, otherwise, a `cancelor` function that handles operation cancelling is returned.

```
/*RTC.js*/
exports._createOffer = function(pc) {
    return function(error, success) {
    pc.createOffer()
        .then(function(offer) {
            success(new RTCSessionDescription(offer))();
        })
        .catch(error);
        return function(a, b, cancelerSuccess) {
           cancellerSuccess(); };
    };
};

exports._createAnswer = function(pc) {
    return function(error, success) {
        pc.createAnswer(success, error);
        return function(a, b, cancelerSuccess) {
           cancellerSuccess(); };
    };
};
```

Figure 5.7: Javascript Foreign Module

In writing the Purescript foregin declaration module shown in Figure 5.8, the tricky part is giving both Javascript functions types that align with their asynchronous nature. `Aff` is a monad for wrapping asynchronous computation in Purescript.
However, we could not directly give the type `RTCPeerConnection -> Aff RTCSessionDescription` to both functions, because `Aff` values could not be directly constructed from FFI definitions in Javascript due to the runtime representation of `Aff`. To do this, we first need to wrap the return values inside `EffectFnAff`, which encapsulates the asynchronous behaviour in Javascript, in the foreign declared functions `_createOffer` and `_createAnswer`, and apply `fromEffectFnAff` to the imported functions to retrieve the desired function type `RTCPeerConnection -> Aff RTCSessionDescription`. `fromEffectFnAff` is a function that turns asynchronous JavaScript behavior into an `Aff` in the PureScript code.

```
/*RTC.purs*/
foreign import _createOffer
  :: RTCPeerConnection -> EffectFnAff RTCSessionDescription

createOffer :: RTCPeerConnection -> Aff RTCSessionDescription
createOffer = fromEffectFnAff <<< _createOffer

foreign import _createAnswer
  :: RTCPeerConnection -> EffectFnAff RTCSessionDescription

createAnswer :: RTCPeerConnection -> Aff RTCSessionDescription
createAnswer = fromEffectFnAff <<< _createAnswer
```

Figure 5.8: Foreign Declarations in Purescript

5.3.3 SetLocalDescription and SetRemoteDescription API

`setLocalDescription` and `setRemoteDescription` also belong to the `RTCPeerConnection` interface. According to Figure 6.2, `setLocalDescription` is called to set the session description of the local end of the connection, while `setRemoteDescription` is to set that of the remote end of the connection, during the offer/answer negotiation process.

- `setLocalDescription()`: asynchronously alters the local session description of the connection to the session description passed as a parameter, and returns a `Promise` which is resolved when the description has been modified [23].
- `setRemoteDescription()`: asynchronously sets the session description of the remote end of the connection as the specified session description, the remote peer's current offer or answer. Similar to `setLocalDescription`, its return values is a `Promise`, which is fulfilled if the session description is altered successfully [32].

```
/*setLocalDescription*/
offer = Alice.createOffer()
ldPromise = Alice.setLocalDescription(offer)
/*setRemoteDescription*/
answer = Bob.createAnswer()
rdPromise = Alice.setRemoteDescription(answer)
```

Figure 5.9: Example Usage of the `setLocalDescription` and `setRemoteDescription` API

The definitions of the Javascript foreign modules in Figure 5.10 for these two functions are very similar to that of `createOffer` and `createAnswer`, since both `setLocalDescription` and `setRemoteDescription` are asynchronous as well. An

asynchronous function containing success and error handling callbacks is returned in the inner function and is propagated to the top-pest level function as the return value. The APIs `setLocalDescription` and `setRemoteDescription` are called upon `pc`, a `RTCPeerConnection`, inside the return asynchronous function. Again, a `cancelor` function is returned when the runtime wants to cancel the operation, also, the new definitions are named as `_setLocalDescription` and `_setRemoteDescription`, the presence of the `_` prefix is again to avoid confusion.

Since the `setLocalDescription` and `setRemoteDescription` API receive multiple arguments, `RTCSessionDescription` and `RTCPeerConnection`, whilst Purescript's function types only take a single parameter at a time, we could not just define their foregin Javascript functions as normal Javascript functions with multiple parameters. We need a way to simulate Purescript functions and one of the approaches is via *currying*, even though there are shortcomings concerning performance. Therefore, from Figure 5.10, `_setLocalDEscription` is declared as a function that takes a single argument `RTCSessionDescription`, which returns a single-argument function that takes the second argument, `RTCPeerConnection`. This way it perfectly matches the currying behaviour of Purescript's functions.

```
/*RTC.js*/
exports._setLocalDescription = function(desc) { return function(
   pc) {
    return function(error, success) {
        pc.setLocalDescription(desc, success, error);
        return function(a, b, cancelerSuccess) { cancellerSuccess
           (); };
    };
};};

exports._setRemoteDescription = function(desc) { return function(
   pc) {
    return function(error, success) {
        pc.setRemoteDescription(desc, success, error);
        return function(a, b, cancelerSuccess) { cancellerSuccess
           (); };
    };
};};
```

Figure 5.10: Javascript Foregin Module

Once the Javascript foreign module is constructed, the next step is giving them Purescript function types. Both `_setLocalDescription` and `_setRemoteDescription` are given type `RTCSessionDescription -> RTCPeerConnection -> EffectFnAff Unit`. Similarly, since the Javascript asynchronous behaviour could not be directly converted into the asynchrounous representation `Aff` in Purescript, we again wrap the return value `Unit` inside `EffectFnAff`, which could be turned into `Aff` by applying `fromEffectFnAff`. We are using `Unit` as the return type because we want to ig-

nore the session description, which has no usage in our work flow, returned by the Javascript API.

```
/*RTC.purs*/
foreign import _setLocalDescription
  :: RTCSessionDescription -> RTCPeerConnection -> EffectFnAff Unit

setLocalDescription :: RTCSessionDescription -> RTCPeerConnection -> Aff Unit
setLocalDescription desc pc = fromEffectFnAff $ _setLocalDescription desc pc

foreign import _setRemoteDescription
  :: RTCSessionDescription -> RTCPeerConnection -> EffectFnAff Unit

setRemoteDescription :: RTCSessionDescription -> RTCPeerConnection -> Aff Unit
setRemoteDescription desc pc = fromEffectFnAff $ _setRemoteDescription desc pc
```

Figure 5.11: Foreign Declaration in Purescript

5.3.4 CreateDataChannel API

The `createDataChannel` interface allows a data channel to be added to a connection for peers to send and receive media data over the connection. This data channel allows secure exchange of arbitrary data, simply any type of data. `createDataChannel` is usually called by the connection initiator, then, the remote peer will be noticed of the created data channel by listening on the remote end of the connection, and gain access to the data channel. The below shows an example of adding a data channel to a connection using the API [24].

```
/*On the local end*/
Alice = new RTCPeerConnection(..);
let dataChannel = Alice.createDataChannel("Data Channel");

/*On the remote end*/
Bob = new RTCPeerConnection(..);
Bob.ondatachannel = function(event) { receiveChannel = event.
   channel;}
```

Figure 5.12: Example Usage of the `createDataChannel` API

In the above example, both `Alice` and `Bob` are newly created `RTCPeerConnection` objects. `Alice` as the connection initiator calls `createDataChannel()` on itself and obtains a `DataChannel` object. On the other end of the connection, `Bob` assigns an event handling function to `ondatachannel`, a property of a `RTCPeerConnection` object, which allows developers listen for any newly created data channels on the

opposite end. The parameter passed is a label tag for the created data channel, and will be shown in the data channel metadata.

Like all the Javascript foreign modules declared in the above, a foreign function `createDataChannel` in Figure 5.13 is defined and assigned to `exports`. It also simulates the *currying* of Purescript's functions by creating a chain of nested single-argument function calls. `createDataChannel` first takes in a `String`, the tag of the data channel, then returns a function that takes a `RTCPeerConnection`, and evetually returns a no-argument function that calls the API `createDataChannel` on the input connection object. Returning a function with zero arguments is to match the `Effect a` return type in Purescript, which delays the execution of the body until `createDataChannel` is called.

The right of Figure 5.13 displays the function type declaration for the Javascript foreign function `createDataChannel`, which is `String -> RTCPeerConnection -> Effect RTCDataChannel`, where `String` corresponding to the data channel label and `RTCPeerConnection` corresponding to a webrtc connection. The return data channel is wrapped inside an `Effect` monad, for the purpose which has explained earlier.

```
/*RTC.js*/
exports.createDataChannel =
  function(s) {
     return function(pc) {
         return function() {
             var dc = pc.
                createDataChannel
                (s);
             return dc;
         };
     };
};
```

```
/*RTC.purs*/
foreign import data
  RTCDataChannel :: Type
foreign import
   createDataChannel
  :: String ->
     RTCPeerConnection ->
     Effect RTCDataChannel
```

Figure 5.13: Javascript Foreign Module(Left), Foreign Declaration in Purescript(Right)

To avoid repetitive explanations, only the key Purescript bindings are shown in this section. All the other not yet mentioned Purescript bindings are attached to the appendix.

5.4 Implementation

Having completed the Purescript bindings for the WebRTC Javascript API, the following step is to incorporate them into the **Purescript Session Runtime Library**. As mentioned in section 4.3.2, the transport layer currently provides communication options including `WebSockets` and `Memory`, the task here is to add a third option `WebRTC`, allowing library users to establish a direct client-to-client session with other clients.

5.4.1 Overview of Transport Type Class

The transport abstraction is implemented by the `Transport` type class in the library, with the definition shown in Figure 5.14. Based on its definition, we see that any instances of the `Transport` type class need to provide support for `send`, `receive` and `close`, which are the basic primitives that any entity in a protocol has to be equipped with. More specifically, if the role of an entity is a client, it also needs to implement a `connect` method, which we can see from the `TransportClient` type class extended from `Transport`. Likewise for entity who is a server, a `TransportServer` is declared and extended from `Transport`, stating that it must have defined a `serve` method for serving the requests from clients.

The type variable `c` represents the transport medium, for instance `WebSockets`, whilst type variable `p` represents the type of the path the transport body is connecting to, for example `URL`. To be able to

As a result, to provide a WebRTC option for library users, we need to (1) implement `send`, `receive`, `close`, `connect` and `serve` for WebRTC (2) instantiate instances of the `Transport` type class, `TransportClient` type class and `TransportServer` type class (3) modify the type classes' definitions to

```
class Transport c p | c -> p where
  send    :: forall m. MonadAff m => c -> Json -> m Unit
  receive :: forall m. MonadAff m => c -> m Json
  close   :: forall m. MonadAff m => c -> m Unit

class Transport c p <= TransportClient c p | c -> p x where
  connect :: forall m. MonadAff m => p -> m c

class Transport c p <= TransportServer c p | c -> p x where
  serve :: forall m. MonadAff m => p -> m c
```

Figure 5.14: Definition of `Transport` Type Class

5.4.2 WebRTC Connect and Serve

Challenges

Implementing `connect` and `serve` is the trickiest part of all, not only because creating a WebRTC connection involves a lot of message exchange through the signalling server between clients, and also the amount of information provided by the library user is restricted by the below abstract definitions given by the type classes.

```
connect :: forall m. MonadAff m => p -> m c
serve :: forall m. MonadAff m => p -> m c
```

Figure 5.15: The Abstract Definitions of `connect` and `serve`

According to the above function types, both `connect` and `serve` should only accept a single argument which is a path of any type. Considering `connect` in the case of WebSockets, the path is going to be the address of the server for a client to connect to, and this is simply sufficient information for a client to create a WebSocket connection with a server. In the case of WebRTC, a client needs to connect to a signalling server first, in order to connect to a remote client, therefore the path needs to be the address of the signalling server.

However, given that the role of a signalling server is to relay information between peers when they are establishing a WebRTC connection, there probably will be other information the library user wants to provide to the signalling server, for example, the ID of the client, the ID of the remote client it wants to connect to, and etc, depending on the implementation of the signalling server and the nature of the web application. Therefore, this shows that the abstract definitions of these two functions do not cover the case of WebRTC, and need to be modified.

To allow an arbitrary amount of extra information to be given to the abstract functions `connect` and `serve`. From Figure 5.16, we can see that a type parameter `x` is added to `TransportClient` and `TransportServer`. The type parameter `x` belongs to *kind* `# Type`, where *kind* could be known as the type of a type in Purescript. `# Type` is a type-level representation of a row of types, meaning that library users can construct a customised row of data types for holding the extra information they want to send to the target pointed by the path `p`. In the abstract definitions of `connect` and `serve`, a parameter of type `Record x` is added instead of `x`, because function types only accept parameters of *kind* `Type`, therefore, `Record`, a type constructor with the definition `# Type -> Type`, is applied to `x` to convert it into a type of *kind* `Type`.

```
class Transport c p | c -> p where
  send    :: forall m. MonadAff m => c -> Json -> m Unit
  receive :: forall m. MonadAff m => c -> m Json
  close   :: forall m. MonadAff m => c -> m Unit

class Transport c p <= TransportClient c p (x :: # Type) | c ->
    p x where
  connect :: forall m. MonadAff m => p -> Record x -> m c

class Transport c p <= TransportServer c p (x :: # Type) | c ->
    p x where
  serve :: forall m. MonadAff m => p -> Record x -> m c
```

Figure 5.16: Modified Definition of `Transport` Type Class

Files and Imports

`connect` and `serve` of WebRTC are implemented in a file named `WebRTC.purs`, which imported all the Purescript bindings of the WebRTC Javascript API needed for the following implementations in the file `RTC.purs`.

WebRTC Connect

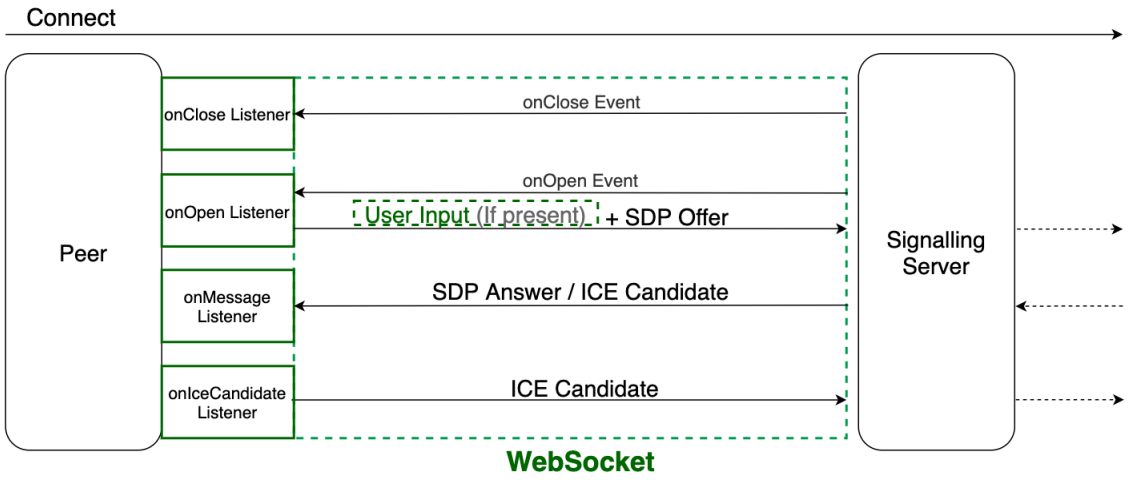


Figure 5.17: Overview of the Implementation of `connect`

A WebRTC's `connect` initiates the **SDP offer/answer negotiation** with a remote peer, undergoes **ICE candidates exchange**, and eventually returns a WebRTC connection object `WebRTCConnection` to the client if successful. Here is the declaration of the data constructor `WebRTCConnection`.

```
data WebRTCConnection =
  WebRTCConnection RTCDataChannel RTCPeerConnection (AVar DataChanStatus) (AVar Json)
```

Figure 5.18: Definition of the data constructor `WebRTCConnection`

`AVar` is an asynchronous variable constructor, which takes a `Type` and returns an asynchronous variable of that type [29]. `AVar DataChanStatus` stores the data channel's status of a connected WebRTC connection. `AVar Json` stores data received from the data channel.

As `connect` is called, an async `RTCPeerConnection` object is created via `newRTCPeerConnection`. Next, a WebSockets channel is created asynchronously to connect to the signalling server. Figure 5.17 depicts an overview of different types of message exchanging between the signalling server and the client via the WebSockets channel.

Once a web socket is created, three types of listeners are added to it, which are as follows.

- **onOpen Listener**: Listens for events indicating that the web socket channel is opened successfully. According to Figure 5.17, once the web socket is opened, if the library user has passed a custom message as a parameter to `connect`, it will first be sent to the signalling server. Then a SDP offer is created via `createOffer` and is also sent to the signalling server. After that, on the client side, `setLocalDescription` is called to set the local session description as the newly created offer. The following is a fragment of the code that is invoked by the listener.

```
sendOffer ::
  RTCPeerConnection -> WS.WebSocket -> RTCConnectionInfo -> Aff Unit
sendOffer conn socket connInfo = do
  offer <- createOffer conn
  offerStr <- createOfferString offer
  msgStr <- createSDPMessageString
              (SDPMessage {"type":"offer",
                           "sender": connInfo.thisPeerId,
                           "receipient": connInfo.remotePeerId,
                           ...,
                           "content": offerStr})
  liftEffect $ WS.sendString socket msgStr
  setLocalDescription offer conn
```

Figure 5.19: Code Snippet of Handling WebSockets `onOpen` Event

- **onMessage Listener**: Listens for SDP answers relayed by the signalling server. If a SDP answer is received, `setRemoteDescription` is called to change the remote session description of the connection to the received answer. The below shows the code the handles a SDP answer. It also listens for ICE candidates sent

```
handleAnswer :: RTCPeerConnection -> RTCSessionDescription -> Aff
   Unit
handleAnswer conn answer = do
  answerStr <- createAnswerString answer
  liftEffect $ log ("Received answer: " <> answerStr)
  setRemoteDescription answer conn
```

Figure 5.20: Code Snippet of Handling WebSockets `onMessage` Event

  by the remote peer via the signalling server. If a ICE candidate is received, the ICE candidate is added to our end of the connection via `addIceCandidate`.

- **onClose Listener**: Listens for events indicating that the web socket channel is closed. If such event arrives, we close the web socket channel on our end.

Apart from adding listeners to the web socket channel that connects to the signalling server, a listener for gathering ICE candidates proposed by our end is added to the `RTCPeerConnection`. Once there are new proposals for ICE candidates, an event is sent to this listener and the listener invokes the function that sends the newly proposed candidates to the remote peer via the web socket channel. A fragment of the code is attached in the following.

```
connect ... = do
  ...
  liftEffect $ onicecandidate (onIceCandidateHandler socket pInfo) conn
  ...

onIceCandidateHandler ::
  WS.WebSocket -> RTCConnectionInfo -> (IceEvent -> Effect Unit)
onIceCandidateHandler socket connInfo = \ev -> launchAff_ $ do
  case (iceEventCandidate ev) of
    Nothing -> pure unit
    Just c -> do
      msgStr <- createSDPMessageString
                  (SDPMessage {"type": "ice",
                               "sender": connInfo.thisPeerId,
                               "receipient": connInfo.remotePeerId,
                                .. ,
                               "content": (stringify $ encodeJson c)})
      liftEffect $ WS.sendString socket msgStr
```

Figure 5.21: Code Snippet of Handling `onIceCandidate` Event

To be able to send arbitrary data over the connection, `createDataChannel` is called to create a data channel. Listeners including `onopen`, `onmessage` and `onclose` are added to the newly created data channel. Particularly, `onmessage` listens for arrival of data over the data channel, any received data is put into a buffer which is also returned together with the `RTCPeerConnection` at the end of this function.

To ensure the WebRTC connection is returned only if the connection is well established, an `onconnectionstatechange` listener is added to `RTCPeerConnection` to observe the change of state of the connection, this way, the WebRTC connection is returned only when the state is `connected`, indicating that the connection is connected to the remote peer successfully.

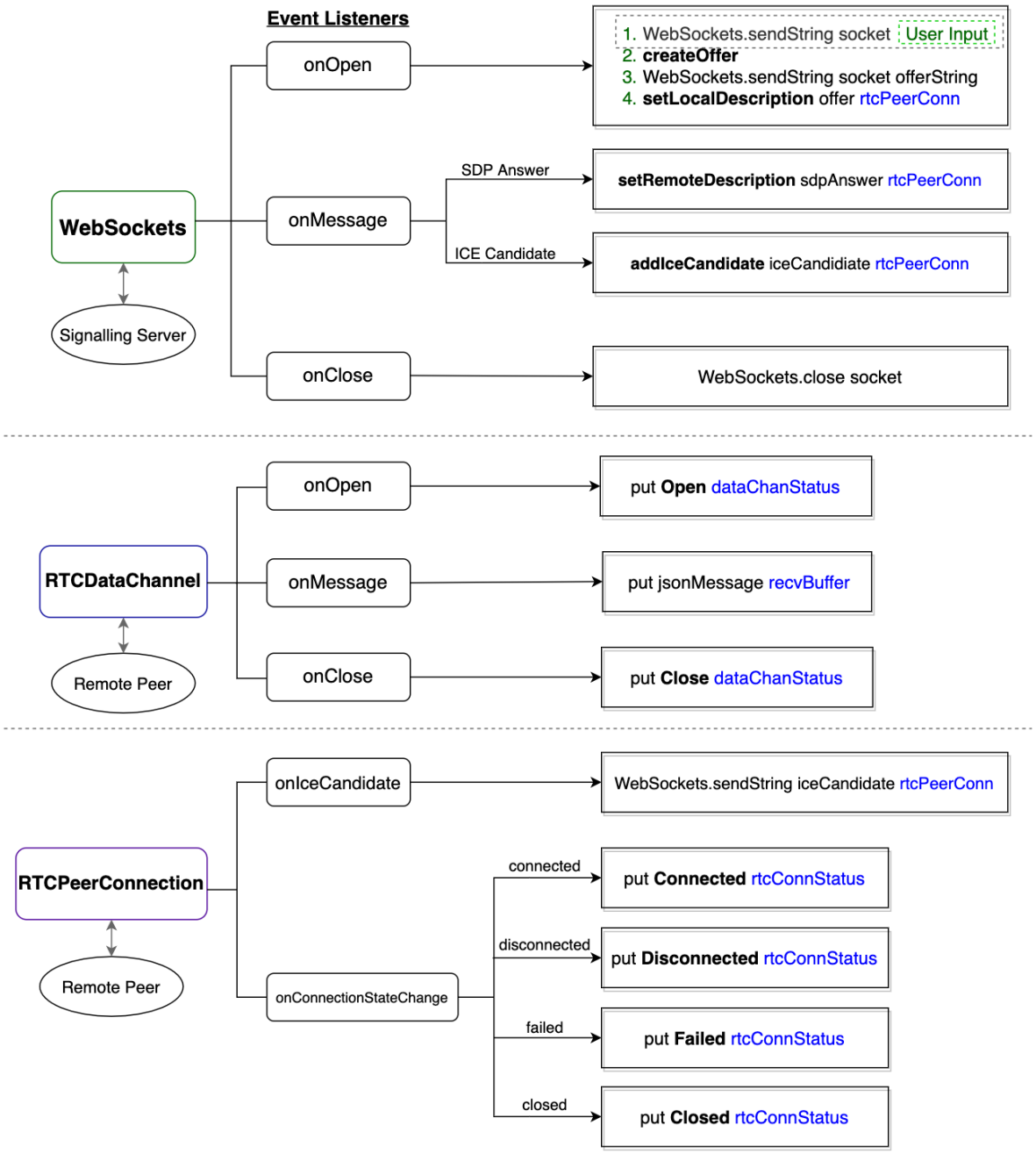


Figure 5.22: Overview of Event Listeners in `serve`

WebRTC Serve

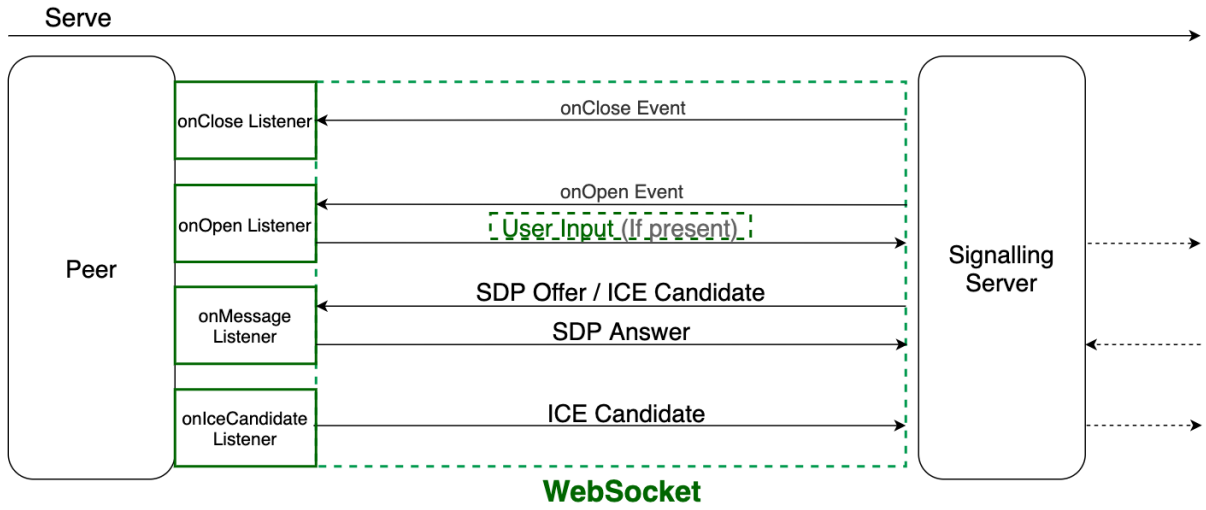


Figure 5.23: Overview of WebSockets Data Exchange in `serve`

Different from `connect`, `serve` waits for an arrival of a SDP offer, and subsequently participates in a **SDP offer/answer negotiation** and a **ICE candidate negotiation** with a remote peer through a signalling server. Same as **connect**, it returns a WebRTC connection object `WebRTCConnection` (which is defined in Figure 5.18) to the client if agreements have been achieved.

Similar to `connect`, as `serve` is performed, a RTCPeerConnectionobject is created via `newRTCPeerConnection`, and a WebSocket channel is created to connect to the signalling server. Both are done asynchronously. The rest of the execution flow of `serve` is very much the same as `conneci`.

The only difference is what happens during the **SDP offer/answer negotiation**, which takes place between the client and a signalling server at the WebSockets channel. With reference to Figure 5.23, the client does not send anything to the signalling server unless the user needs to submit a custom message to the server before the negotiation starts. If a SDP offer arrives, this indicates that there exists a remote peer who wants to establish a WebRTC connection, and the client needs to respond with a SDP answer. These actions are triggered by the event listeners attached to the web socket object, which are listed in the below.

- **onOpen Listener:** Called when the web socket connection is opened. If a user-defined input is passed to `serve`, it sends that to the signalling server.
- **onMessage Listener:** Listens for SDP offer sent by the remote peer via the signalling server. If an offer is sent over, `setRemoteDescription` is called to set the remote session description of the connection to the received offer, then, create a SDP answer via `createAnswer` and send the answer back to the remote peer through the web socket channel via the signalling server. Finally, call `setLocalDescription` to modify the session description of the local end of the connection to the newly created answer.

```
sendAnswer ::RTCPeerConnection -> WS.WebSocket -> RTCConnectionInfo
   -> RTCSessionDescription -> Aff Unit
sendAnswer conn socket connInfo sdp = do
  setRemoteDescription sdp conn
  answer <- createAnswer conn
  answerStr <- createAnswerString answer
  setLocalDescription answer conn
  answerStr <- createSDPMessageString
                (SDPMessage {"type": "answer",
                             "sender": connInfo.thisPeerId,
                             "receipient": connInfo.remotePeerId,
                             ...,
                             "content": answerStr})
  liftEffect $ WS.sendString socket answerStr
```

Figure 5.24: Definition of the function called as a SDP offer arrives

Figure 5.25 displays all the listeners that are attached to the web socket channel, the WebRTC data channel and the WebRTC connection, and their associated methods.

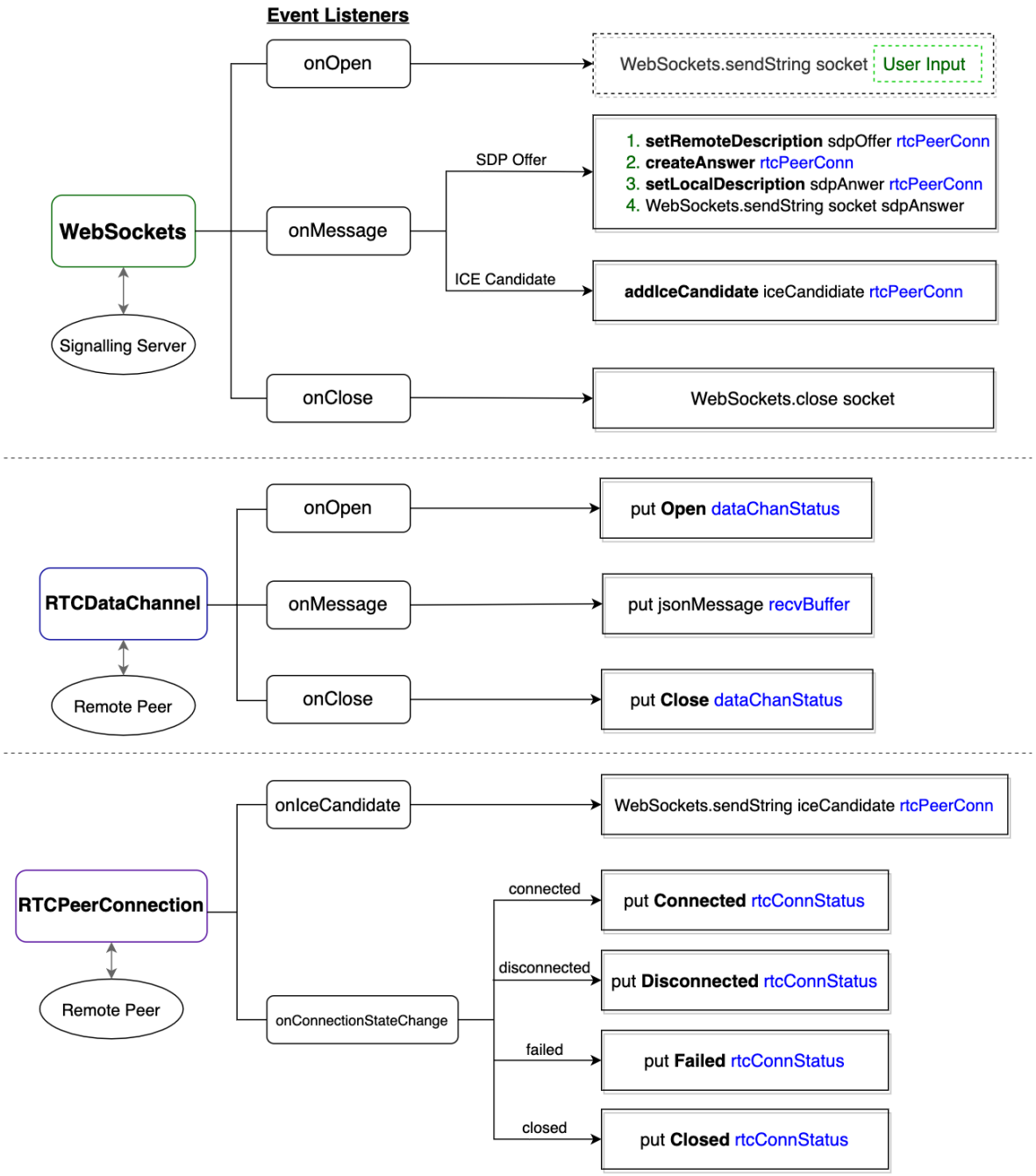


Figure 5.25: Overview of Event Listeners in `connect`

5.4.3 WebRTC Send

A WebRTC's `send` sends arbitrary data over the connected WebRTC connection to a remote peer. From Figure 5.26, `send` takes a `WebRTCConnection` object and data to be sent of type `Json`. Before sending, we check whether the data channel embedded in the `WebRTCConnection` object is opened or not. If it is opened, we convert the input data from `Json` to `String`, and sends it over the data channel, denoted by `dc` in the figure. Otherwise, nothing is performed.

```
send :: WebRTCConnection -> Json -> Aff Unit
send (WebRTCConnection dc _ sv _) msg = do
  status <- read sv
  case status of
   Open -> liftEffect $ RTC.send (stringify msg) dc
   Closed -> pure unit
```

Figure 5.26: Definition of WebRTC `send`

5.4.4 WebRTC Receive

A WebRTC's `receive` reads from the buffer and returns the data stored if the buffer is not empty. The definition of `receive` is displayed in Figure 5.27. The method only takes a single parameter which is a WebRTC connection. If the data channel is opened, we return what is inside the buffer. If the data channel is closed, we assume that there might be still some unprocessed input, therefore, we attempt to obtain it from the buffer, and return it if present. An error is thrown to notify users that the data channel is closed if the buffer is empty.

```
receive :: WebRTCConnection -> Aff Json
receive (WebRTCConnection _ _ sv buf) = do
  status <- read sv
  case status of
    Open -> take buf
    Closed -> do
      x <- tryTake buf
      case x of
        Nothing -> throwError $ error "Channel is closed"
        Just val -> pure val
```

Figure 5.27: Definition of WebRTC `receive`

5.4.5 WebRTC Close

A WebRTC's `close` simply closes a WebRTC connection and sets the status of the data channel as `Closed`. Once a connection is closed, all ongoing processes, for example, its data channel, ICE candidates processing and etc. are terminated. Additionally, all references to its resources become invalid. The following Figure 5.28 shows its definition.

```
close :: WebRTCConnection -> Aff Unit
close (WebRTCConnection _ pc sv _) = do
  liftEffect $ closeRTCPeerConnection pc
  put Closed sv
```

Figure 5.28: Definition of WebRTC `close`

5.4.6 Instantiation of WebRTC Instances

Once the functions required by the definitions of the type class `Transport`, `TransportClient` and `TransportServer` are in place, the final step is to instantiate the instances of these type classes for a connection type of `WebRTCConnection` with an address of type `URL`, as shown in Figure 5.29.

```
instance webRTCURLTransport :: Transport WebRTCConnection URL where
  send = \ws -> liftAff <<< (send ws)
  receive = liftAff <<< receive
  close = liftAff <<< close

instance webRTCURLTransportClient :: TransportClient WebRTCConnection
    URL ("loginMsg" :: Maybe String, "connInfo" :: {thisPeerId ::
   String, remotePeerId :: String}) where
  connect p x = liftAff $ connect p x.loginMsg x.connInfo

instance webRTCURLTransportServe :: TransportServer WebRTCConnection
   URL ("loginMsg" :: Maybe String, "connInfo" :: {thisPeerId ::
   String, remotePeerId :: String}) where
  serve p x = liftAff $ serve p x.loginMsg x.connInfo
```

Figure 5.29: Declaration of the Instances of WebRTC

5.5 Example Application

In this section, we are going to walk through an implementation of a very simple Purescript application where a peer keeps sending a message to another peer over a WebRTC connection. We demonstrated in the following an end-to-end process, which starts from specifying a protocol in Scribble to writing Purescript functions for each of the roles.

Step 1: Global Protocol Specification

We first created a global protocol in Scribble, shown in Figure 5.30. We see that there are only two roles in the protocol: `ClientA` and `ClientB`, where `ClientA` connects to `ClientB`, then once a connection is established between them, `ClientA` can choose to send `ClientB` a message of type `String` or a `Quit` message to terminate the session.

```
1  module Relay
2
3  type <purescript> "String" from "Prim" as String;
4
5  global protocol Relay(role ClientA, role ClientB)
6  {
7      Connect() connect ClientA to ClientB;
8      choice at ClientA {
9          Message(String) from ClientA to ClientB;
10     } or {
11         Quit() from ClientA to ClientB;
12         disconnect ClientA and ClientB;
13     }
14 }
```

Figure 5.30: Scribble Global Protocol

Step 2: Purescript Implementation of Clients

Given the above Scribble protocol and the code generated from Scribble (attached in Appendices), we wrote a `clientA` function and a `clientB` function on behalves of `ClientA` and ClientB respectively, which are shown in the below.

ClientA

As shown by Figure 5.31, `clientA` creates a `WebRTCConnection` session. Inside the session, it calls `connect` by passing the role it is going to connect with, and a `URL` that contains the address of the signalling server `ws://localhost:80`. What follows

after `connect` are the actions `ClientA` is going to perform over the WebRTC channel once a WebRTC connection is connected with `ClientB`.

```
clientA :: Widget HTML Unit
clientA = session
  (Proxy :: Proxy WebRTCConnection)
  (Role :: Role P2P.ClientA) $ do
    connect (Role :: Role P2P.ClientB) (URL $ "ws://localhost:80") {"loginMsg
        " : (Just $ (stringify <<< encodeJson) {type: "connect", from: "Alice
        ", to: "Bob"}), "connInfo":{thisPeerId: "Alice", remotePeerId: "Bob"
        }}
    select (SProxy :: SProxy "message")
    send (P2P.Message "hello")
    keepSending
    select (SProxy :: SProxy "quit")
    send (P2P.Quit)
    disconnect (Role :: Role P2P.ClientB)
    pure unit
    where
      bind = ibind
      pure = ipure
      discard = bind
      keepSending :: Session (Widget HTML) WebRTCConnection P2P.S13 P2P.S13
          Unit
      keepSending = do
        select (SProxy :: SProxy "message")
        send (P2P.Message "hello")
        keepSending
```

Figure 5.31: Purescript endpoint implementation for `ClientA`

ClientB

Similar to `ClientA`, depicted in Figure 5.32, `clientB` also creates a `WebRTCConnection` session. Then, it calls `serve` by passing the role it is going to be connected with, and a `URL` that contains the address of the signalling server `ws://localhost:80`. again, what follows after `serve` are the actions `ClientB` is going to perform over the WebRTC channel once a WebRTC connection is connected with `ClientA`.

```
clientB :: Widget HTML Unit
clientB = session
  (Proxy :: Proxy WebRTCConnection)
  (Role :: Role P2P.ClientB) $ do
    request (Role :: Role P2P.ClientA) (URL $ "ws://localhost:80") {"loginMsg
        " : (Just $ (stringify <<< encodeJson) {type: "connect", from: "Bob",
         to: "Alice"}), "connInfo":{thisPeerId: "Bob", remotePeerId: "Alice"
        }}
    serving
    pure unit
    where
      serving = choice
        { message: do
          _ <- receive
          lift $ D.text "yo"
          serving
        , quit: do
          void receive
          disconnect (Role :: Role P2P.ClientA)
        }
      bind = ibind
      pure = ipure
      discard = bind
```

Figure 5.32: Purescript endpoint implementation for `ClientB`

5.5.1 Step 3: Signalling Server Implementation

The third step is to build a signalling server that relays the necessary data for creating a WebRTC connection between the two clients. The following Figure 5.33 displays some of the code of the implementation of a node server serving as a signalling server between `ClientA` and `ClientB`.

From Figure 5.33, we see that the node server operates at port 80, and it invokes different functions corresponding to the `type` field of the message sent to it.

```
const WebSocketServer = require('ws').Server
const wss = new WebSocketServer({ port: 80 })

let connections = []

wss.on('connection', (connection) => {
  console.log(process.env.port)

  connection.on('message', (message) => {

    message = parseMessage(message)

    if ( message ) {

      switch (message.type) {
        case 'connect': {
          onConnect(connection, message)
          break;
        }
        case 'offer': {
          onOffer(connection, message)
          break;
        }
        case 'answer': {
         onAnswer(connection, message)
         break;
        }
        case 'ice' :{
          onIceCandidate(connection, message)
          break;
        }
      }

    } else {
      respond(connection, malformedMessage())
    }

  })
```

Figure 5.33: Code fragment of the implementation of the signalling server

5.5.2 Result

Finally, we started the signalling server, and ran `clientB` and `clientA` on different ports on a local machine. We checked the status of the WebRTC connection on `chrome://webrtc-internals`, an internal Chrome tab that holds statistics of any ongoing WebRTC sessions on Chrome. The below Figure 5.34 is a screen capture of the statistics of the WebRTC connection between `clientA` and `clientB`. From the last row of the figure, we can see that the connection state is changed to `connected`, indicating that the connection is created successfully.

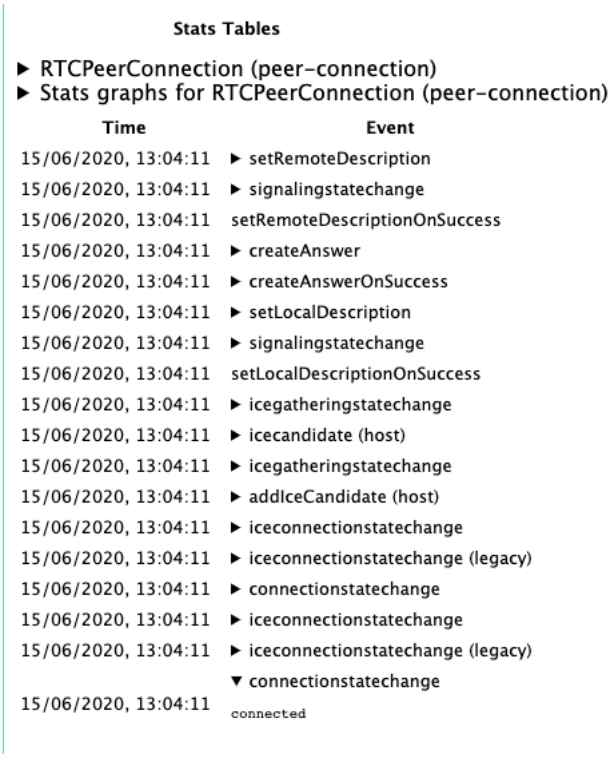


Figure 5.34: Screen Capture of the WebRTC Connection Status

Chapter 6

Benchmark of Purescript Session Runtime Library

6.1 King[31] the Purescript Session Runtime Library

This library is based around type-level programming. It embeds type constraints to its session constructors to check against the generated code, a *static* requirement derived from the EFSM of an input protocol, for protocol violation through constraint solving during compilation. Given that protocol conformance is guaranteed by type checking, we need to assess the extent of increase in the compile time.

King, conducted an evaluation in terms of compilation performance, by implementing several micro-benchmarks to investigate the effect of different features of a protocol on the compile time of an endpoint implementation. To ensure his results are reproducible and detect further undiscovered bugs in this framework, we repeated two of the benchmarks taken by King and implemented two new benchmarks that are broadly used for evaluating session types frameworks.

The implementation of all these benchmarks executes an end-to-end code generation process, going from generating Scribble protocols to implementing one of the roles in the generated protocols. The is done by running a shell script that comprises all stages. Results are then gathered through compiling those functions.

To ensure fairness, all projects were built using `spago build`, with `spago 0.14.0` and `purs 0.13.6`. `spago` is a Purescript package manager and build tool[33], while `purs` is a Purescript compiler[28]. We ran each benchmark 20 times and averaged the results to produce the following graphs.

6.2 Benchmarks

6.2.1 Repeated Benchmarks

Ping-Pong

The Ping-Pong benchmark aims to measure the compile time overhead when there is a long chain of Send and Receive constraints that have to be solved. Repeating King's Ping-Pong benchmark, we generate a set of Ping-Pong protocols, which consists of two roles, with one role sending a Ping message, and the other role responding with a Pong message. The number of Ping-Pong messages sent and received in these generated protocols increases from 2 to 882.

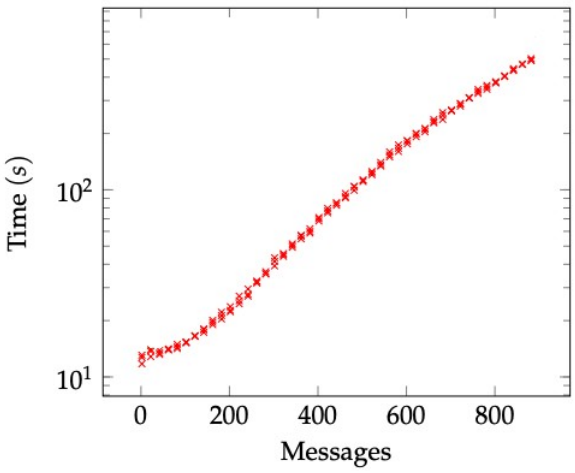
To compare with King's results as fairly as possible, results are collected based on the compilation of the implementation of the role sending a Pong message, which is the same as King's.

```
1 Ping() from A to B;
2 Pong() from B to A;
3 ...
```

```
1 _ <- receive
2 send Pong
3 ...
```

Figure 6.1: Scribble Protocol Structure (Left), Purescript Code Structure (Right)

King's results



As shown by the graph in Figure 6.2, the compile time appears to grow exponentially as the number of Ping-Pong messages increases.

Figure 6.2: King's Results of Ping-Pong[30]

Our results

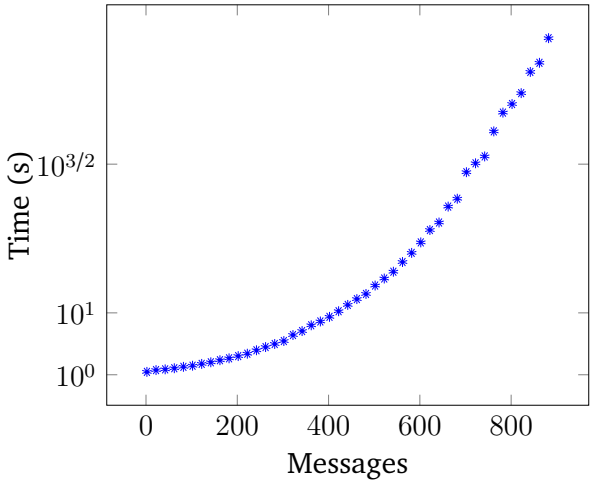


Figure 6.3: Our Results of Ping-Pong

Similar to the results in Figure 6.2, our results also share an exponential complexity. However, the compile time in overall is much lower, as it takes less than a minute when the number of messages reach over 800, whereas from his results, it takes way more than a minute. This could be caused by factors including the choice of build tool, the machine running the benchmark and the implementation of the functions.

Branching

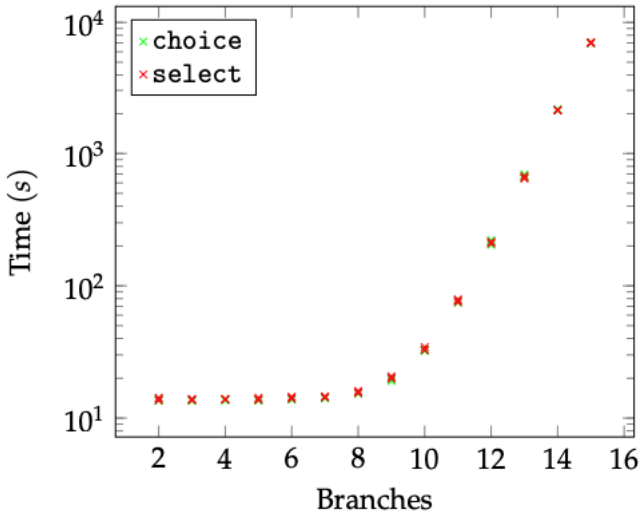
This benchmark targets the branching feature of a protocol. Solving a `Branch` constraint requires parsing of branch options which theoretically should add more compile time overhead compared to `Send` and `Receive`. Taking the same approach as King's, we generate a protocol with n branches, which one of them is selected and followed by a message `send`.

```
choice at A {
  B1() from A to B;
} or {
  Bi () from A to B;
} or ...
```

```
do
  select (SProxy :: SProxy "b1")
  send B1
```

Figure 6.4: Scribble Protocol Structure (Left), Purescript Code Structure (Right)

King's results



From King's results, we see that as the number of branches increases, the compilation time grows super-exponentially. It becomes obviously noticeable when the number of branches reaches 7. This was very worrying as it took more than half an hour to compile for 15 branches, making this library less of a practical use as 15 is not a big number.

Figure 6.5: King's Results of Branching Benchmark[30]

This was a significant issue spotted by King. He discovered that it was caused by a parsing issue in the Purescript's compiler, which occurred when a `RowList` representing branching options was parsed. As the depth of the `RowList` increases, the time needed for parsing increases super-exponentially which was indicated by King's results. The following Figure 6.6 is an example of how branching options are represented using `RowList`.

```
1 (Cons "b1" S5B1 (Cons "b2" S5B2 (... (Cons "bn " S5Bn Nil) ...
    ))
```

Figure 6.6: Example `RowList` of Branch Options

To resolve this problem, we either fix the bug in the compiler or use other alternatives for the branching representation.

Our results

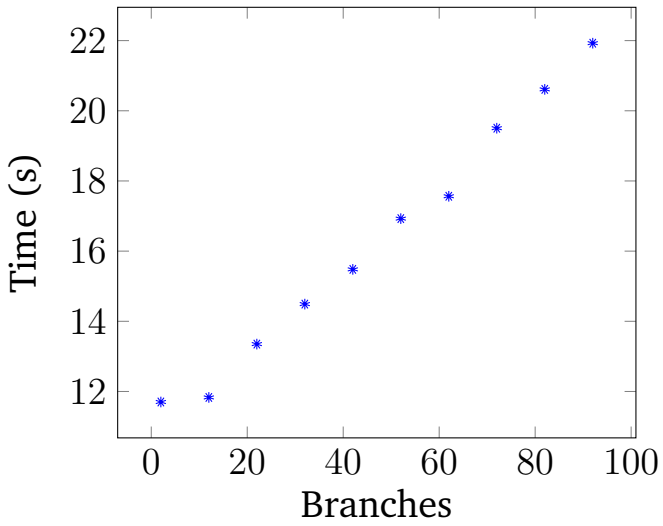


Figure 6.7: Our Results of Branching Benchmark

Referring to the graph in Figure 6.7, it is clear that the issue is no longer present even when the number of branches increases to more than 80. This result is very promising as it takes less than 30 seconds to compile for 92 branches. Moreover, the super-exponential complexity disappears. This was not expected when we took the bench-mark, as no changes were made to the library implementation. Then, we realised that a pull request was submitted to the compiler repository and was resolved recently, eliminating the parsing issue.

6.2.2 New Bench-marks

Ring

A ring protocol resembles a ring network, where every role sends a message to its next adjacent role, then, receives a message from its previous adjacent role. To test if this framework scales properly as the more roles are involved in a ring protocol, we generate a set of ring protocols with increasing number of roles, starting from 3. Here is a small example explaining how this works, if a ring protocol consists of four roles: A, B, C and D. A sends to B, B sends to C, C sends to D, and D sends to A.

Results are collected based on the compilation of the implementation of the first sender in the protocol, which is A in the example just mentioned. In Figure 6.8 which shows our results, we see that the compile time stays relatively at the same level (approximately 2 seconds) as the size of the ring increases. This is reasonable as the amount of constraint solving is constant when the ring varies in size.

```
1  /* Global Protocol */
2  Connect() connect A to B
       ;
3  M() from A to B;
4  Connect() connect B to C
       ;
5  M() from B to C;
6  ...
7  Connect() connect N to A
       ;
8  M() from N to A;
9
10 /* A's Implementation */
11 send Connect()
12 send M()
13 _ <- receive
14 ...
```

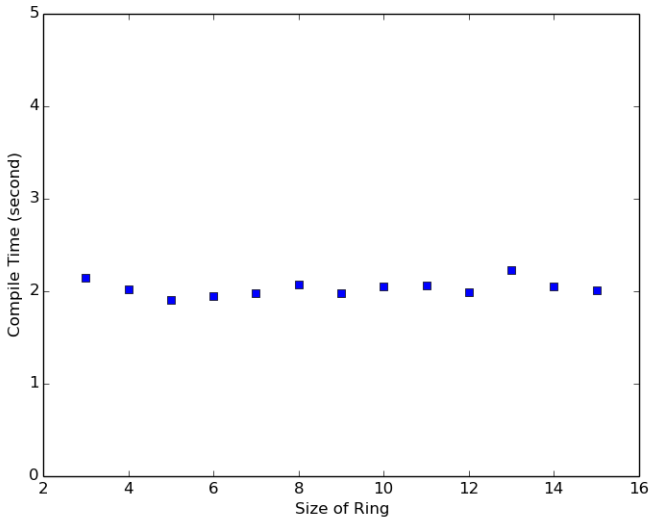


Figure 6.8: Results of Ring Benchmark

One To Many

Here we generate one-to-many protocols with one role sending messages to *n* other roles, with n increases from 1 to 10. The sender role connects to every receiver role and sends a respective message to each of them. Therefore if there are 4 roles in a protocol: A, B, C and D. A connects to B, C and D, and sends a message to B, C, and D sequentially. Again, we measured the compile time based on the sender role, which is A in the example. From our results shown in the following Figure 6.9, the compile time increases gradually as the number of receivers increases. This could be explained by the longer chain of Connect and Send constraints needed to be solved.

```
1  /* Global Protocol */
2  Connect() connect A to B
       ;
3  M1() from A to B;
4  Connect() connect A to C
       ;
5  M2() from A to C;
6  ...
7
8  /* A's Implementation */
9  connect (Role :: Role B)
10 send Connect()
11 send M1()
12 connect (Role :: Role C)
13 send Connect()
14 send M2()
15 ...
```

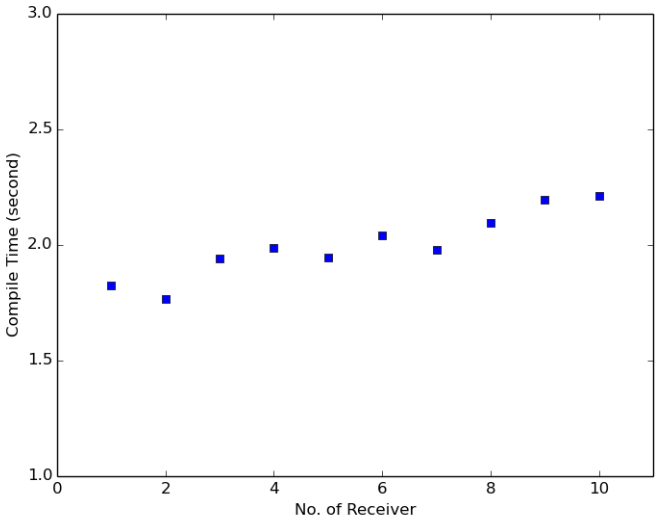


Figure 6.9: Results of One-To-Many Benchmark

Chapter 7

Evaluation

7.1 Improving Error Messages

The error reporting extension generates custom type errors by instantiating instances with a `Fail` type constraint that would match the possible protocol violation errors. We hypothesise that the addition of these type class instances would increase the search space for the constraint solver to solve a particular constraint, increasing the compile time overhead. Combined with the `Fail` type constraint, a further constant compile time cost is added to the resolving of every type class instance. To prove this hypothesis, we performed a set of benchmarks that were previously applied to the library at the same environment, and compare their results.

We have written shell scripts to run an end-to-end code generation process using the error handling extension. To draw a fair comparison, the versions of the compiler and build tool used are identical with what have been used for the Benchmark section in chapter 6, `purs 0.13.6` and `spago 0.14.0`. Also, results are collected by averaging the results of all runs for every bench mark. Again, every benchmark is executed 20 times.

7.1.1 Ping-Pong

A Ping-Pong protocol contains two roles, with one role sending a Ping message, and the other role responding with a Pong message. Here, we generate a set of Ping-Pong protocols with increasing number of messages sent and received, and measure the compile time for each of their corresponding Purescript implementation. The following Figure 7.2 displays our findings.

```
1 Ping() from A to B;
2 Pong() from B to A;
3 ...
```

```
1 _ <- receive
2 send Pong
3 ...
```

Figure 7.1: Scribble Protocol Structure(Left), Purescript Code Structure(Right)

Results

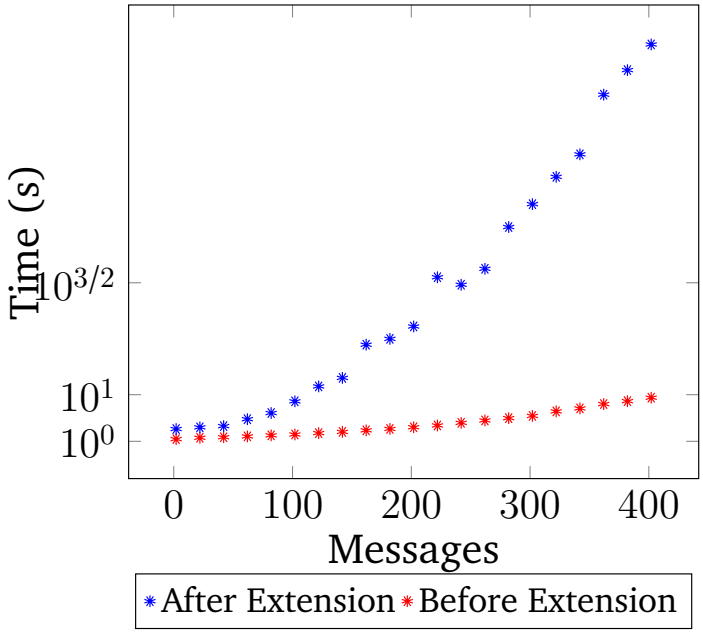


Figure 7.2: Results of Ping-Pong Benchmark

Discussion

According to Figure 7.2, the difference in compile time escalates exponentially as the number of messages increases. When the number of messages reaches around 400, implementation with error handling takes 77 seconds to compile on average, while, implementation without an error handling support only takes approximately 10 seconds. This exponentially increasing difference is caused by the increasing search space for constraint solving, contributed by the increasing number of instances for catching incorrect actions in the generated code, as the length of the chain of `Send` and `Receive` constraints grows. The results are therefore expected from how we implemented the extension.

7.1.2 Branching

While Ping-Pong examines the compile time cost of the extension for a long chain of actions, the Branching benchmark targets the effect of the number of branch options in a branch action.

Again to fairly compare the compilation performance before and after the extension is implemented in the Session Runtime Library, we replicated the implementation of the Branching benchmark in chapter 6, where we generate a set of protocols with increasing number of branch options up to 100. Figure 7.4 shows our results.

```
choice at A {
B1() from A to B;
} or {
Bi () from A to B;
} or ...
```

```
do
select (SProxy :: SProxy
   "b1")
send B1
```

Figure 7.3: Scribble Protocol Structure(Left), Purescript Code Structure(Right)

Results

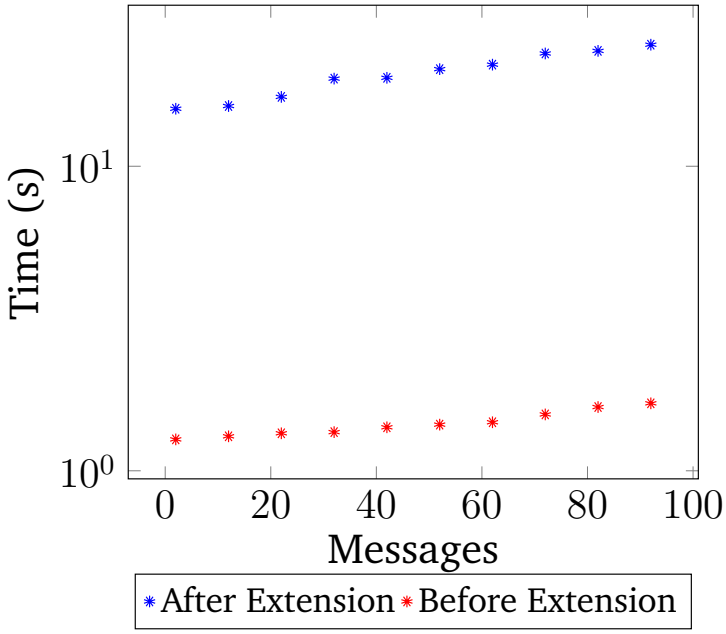


Figure 7.4: Results of Branching Benchmark.

Discussion

From the above Figure 7.4, the compilation time of both follows a similar steady upward trajectory, which is led by the increasing amount of time needed to parse

branch options when resolving the `select` action as the number of branches increases. The upward shifting of the entire curve of that after extension is due to the constant number of *error reporting* instances added for the `send` and `select` action, which constantly increased the search space for the constraint solver and thus the overall compilation time.

7.2 Conclusion

The error reporting extension leads to an increased compilation time cost. Increasing number of branch options only adds a fixed compile time overhead. The compile time cost becomes strikingly huge when there is a long chain of actions, as shown by the results from the Ping-Pong benchmark. This is stemmed from the method we are using to produce custom type errors, therefore, to tackle this issue we need to switch to an entirely new approach. However, although the rate of the increase in compilation time grows exponentially when the number of communication action increases, the actual increase is not significantly problematic from the practical perspective, as it only takes one and a quarter minute to compile an implementation with around 400 communication actions.

7.3 WebRTC Client-to-Client Session

One of the challenges of implementing this WebRTC extension is applying the library's transport abstraction to the WebRTC use case while making sure the extension is applicable to any type of WebRTC applications. During the process, we made a design decision to modify the transport type classes' definitions to cater the need for library users to send custom messages to the signalling server before making a WebRTC connection. We also decided what information is needed from the user to facilitate the WebRTC connection establishment via the signalling server.

Since there are many design choices made concerning the structure of information need to create a WebRTC session, it is important to test whether the design decisions are made correctly to suit a wide spread of use cases. Hence, we consolidated two mainstream WebRTC applications, as shown in the below, and see if our design is capable of implementing each of the use cases.

- **Zoom**, a popular video Streaming platform, where users could a create private room which requires password or a public room that is accessible by anyone. Users have to provide their names and the ID of the room they want to connect to.

- **Facebook Messenger**, an instant messaging application for users to send and receive texts from each other instantly.

7.3.1 Data Structure for Containing the Required Data

As mentioned in the above section, we defined a data structure for containing both the custom data user wants to send to the signalling server and the data we require from user. A representation of the data structure is shown in the following Figure 7.5. **loginMsg** holds the optional custom message, whereas **connectInfo** holds the compulsory information from user: **thisPeerId** and **remotePeerId**, indicating the id of the user and the id of the remote peer the user is connecting to respectively.

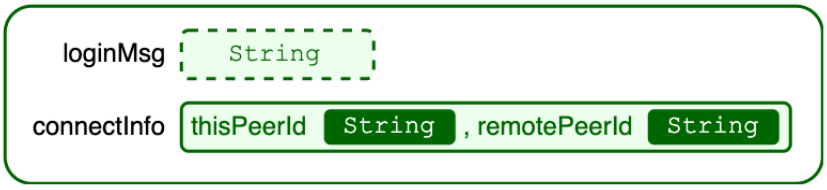


Figure 7.5: Representation of the Data Structure for Holding the Information

7.3.2 Facebook Messenger

Here we implemented a use case when a client is trying to connect to another peer on the messenger to start a conversion. Therefore, it is expected that the user needs to provide his/her id and the id of the person he/she is trying to chat with. The following is the function that implements this use case.

```
1 connectFBMessenger :: String -> String -> Effect Unit
2 connectFBMessenger thisId remoteId  = session
3   (Proxy :: Proxy WebRTCConnection)
4   (Role :: Role Zoom.ThisPeer) $ do
5     connect (Role :: Role Zoom.RemotePeer) (URL $ "..")
6       {"loginMsg": (Just $ (stringify <<< encodeJson)
7          {type: "connect",
8           from: thisId,
9           remote: remoteId}),
10       "connInfo": {thisPeerId: thisId,
11                    remotePeerId: remoteId}}
12    ...
```

Figure 7.6: Code snippet of the implementation of a function connecting to a friend on Facebook Messenger

According to Figure 7.6, we implemented a function for user to connect to a remote friend on the messenger. The function takes two arguments: the user's id and the remote peer's id. Then, we constructed a custom message that wraps these two ids in a `row`, shown from line 7 to line 9, which is then transformed into `String`, and is set as the value of the field `loginMsg` in line 6, aligning with the type definition in Figure 7.5.

7.3.3 Zoom

There two scenarios expected in connecting to an online meeting room when using Zoom: (1) Connecting to a private room where user needs to input a password, ID of the meeting room and the user's name. (2) Connecting to a public room where user only needs to enter the ID of the room and his/her name. The following displays the implementation for each of these scenarios.

Connecting to a private room

```
1 connectToPrivate :: String -> String -> String -> Effect Unit
2 connectToPrivate password id name = session
3   (Proxy :: Proxy WebRTCConnection)
4   (Role :: Role Zoom.ThisPeer) $ do
5     connect (Role :: Role Zoom.RemotePeer) (URL $ "..")
6       {"loginMsg": (Just $ (stringify <<< encodeJson)
7          {type: "connectToPrivateRoom",
8           from: name,
9           roomId: id,
10          pw: password}),
11       "connInfo": {thisPeerId: name,
12                    remotePeerId: id}}
13    ...
```

Figure 7.7: Code snippet of the implementation of a function connecting to a private room in Zoom

As demonstrated in Figure 7.7, we successfully implemented a function for a client to connect to a private room with the requirement to provide a password, room ID and user name to the signalling server before establishing a WebRTC session. From line 6 to line 10, using `row` type in Purescript, we constructed a `row` comprising the user name, room ID and password, and converted it into a `String`, which matches the type expected from a `loginMsg` field with reference to the definition in Figure 7.5.

Connecting to a public room

As demonstrated in Figure 7.8, again we implemented a function for a client to connect to a public room using a room ID and a user name, which are needed by the signalling server before establishing a WebRTC session. From line 7 to line 9, we constructed a custom type which is a `row` comprising the user name and room ID. Then we converted it into a `String`, which again matches the type expected from a `loginMsg` field with reference to the definition in Figure 7.5.

```
connectToPublic :: String -> String -> Effect Unit
connectToPublic id name = session
  (Proxy :: Proxy WebRTCConnection)
  (Role :: Role Zoom.ThisPeer) $ do
    connect (Role :: Role Zoom.RemotePeer) (URL $ "..")
      {"loginMsg": (Just $ (stringify <<< encodeJson)
         {type: "connectToPublicRoom",
          from: name,
          roomId: id}),
       "connInfo": {thisPeerId: name,
                    remotePeerId: id}}
    ...
```

Figure 7.8: Code snippet of the implementation of a function connecting to a public room in Zoom

7.3.4 Conclusion

As a whole, the data structure, defined for transferring the necessary information prior to making a WebRTC connection with a remote peer, could be used to implement two of the most popular use cases in the industry. Hence, we can conclude that our design does provide a certain degree of usability for the library users to build a range of real-time WebRTC applications.

Chapter 8

Conclusion

8.1 Summary

In conclusion, we extended the code generation of Scribble-Purescript to improve the quality of error messages prompted by protocol violations, incorporated WebRTC, a real-time communication API embedded in modern browsers, into the Session Runtime Library, permitting library users to construct client-to-client sessions, and carried out an extended evaluation of the Session Runtime Library.

Improving Error Messages

We identified that type error messages generated from compiling protocol violating endpoint implementation are difficult to read and to understand the cause of error. As the complexity of a protocol grows, the type errors arising from protocol violation also become a pain for debugging. To tackle this issue, we developed a solution to produce error messages that provide more context about the error including the communication action that is against the protocol, the expected communication action to be performed and etc. This is achieved by (1) devising a new structure for the generated Purescript code that is able to embed useful type-level information for error reporting and to trigger the compiler to customise an error message based on those information when a breach of protocol occurs, (2) extending the code generation of Puresciprt-Scribble to produce the new representation. As a result, the generated code has the capability to catch type errors that are caused by actions disobeying the pre-agreed protocol and prompt the corresponding useful information needed to resolve the problem.

Extending Transport Layer: WebRTC Client-to-Client Connection

We extended the Purescript Session Runtime Library to support constructing direct sessions between clients via WebRTC. The original transport options provided by

the library in King's et al[14, 31] are WebSockets and Memory which do not allow browser-to-browser connections. Hence, our library did not work with protocols involving client-to-client communication. With WebRTC gaining traction in bringing much faster and more stable real-time communication through direct communication amongst browsers, we incorporated WebRTC into the library taking advantage of the parameterisation of the transport layer, which enables library users to establish client-to-client sessions in their web applications. Combined with WebRTC, developers could input protocols consisting of client-to-client interactions to Scribble, and use the generated code from Scribble to build real-time web applications via WebRTC in Purescript.

Benchmark of Purescript Session Runtime Library

We re-evaluated the compilation performance of the Purescript Session Runtime Library via conducting multiple benchmarks, which are widely used for assessing multiparty session types implementations. We implemented the benchmarks which are tailored to this framework. This work is also carried out to investigate about the issue of the massive compile time overhead associated with branching in the library, which was discovered previously by Jonathan King, the author of the library. Unexpectedly, the result of our benchmark shows that the issue is no longer present, which is resolved by a recent update of the Purescript compiler. Results obtained from other benchmarks are reasonable, showing that the library could be used for implementing communication protocols to a certain scale and complexity.

Evaluation

We assessed the compilation performance of the error reporting feature by carrying out a set of benchmarks. The results show that adding support for error reporting increases the compilation time cost. There is a surge in compile time compared to that without the extension implemented is particularly severe in scenarios when there is a long chain of communication actions, caused by an excessively large number of *error-reporting* instances to be generated, which sharply increases the time needed for constraint solving, and thus increases the compile time significantly. However, this actual increase is not practically worrying, unless the library is used to an extreme scale. For the WebRTC feature, we tested whether our implementation could be used for various popular use cases for WebRTC. We implemented multiple functions to mimic Zoom, a popular video streaming application, and Facebook Messenger, another well-known texting application. The result of these implementations shows that our design allows users to build WebRTC applications for these purposes and is not applicable only for a narrow range of applications.

8.2 Future Work

Allowing Audio and Video Streaming via WebRTC

The current implementation of WebRTC session only allows arbitrary data to be transmitted. One possible future extension is to extend the WebRTC part of the library to support transfer of audio and video amongst clients. This requires writing Purescript bindings for the WebRTC Javascript API `getUserMedia` for acquiring media streams, and possibly re-implementing the `connect` and `serve` functions or even the transport abstraction to enable library users to select the type of WebRTC stream.

Node-WebRTC for Server Side Implementation

Node-WebRTC [25] is a Javascript bindings to WebRTC provided for Node js. The current implementation for supporting WebRTC only applies to client side implementations in Purescript. Node-WebRTC could be integrated into the Session Runtime Library to provide WebRTC support for any Purescript server side implementations, so that a Web-RTC session could support any types of communication, not just client-to-client.

Investigation about the Functional Dependencies in the Type Classes' Declarations for EFSM Encoding

It is discovered that the functional dependency $s \rightsquigarrow a$ in the type class declarations of `Send` and `Receive` in the Session Runtime Library could be eliminated, because removing it does not give any compile errors for several example applications implemented using the library. More investigations and testing are needed to justify this removal, as this impacts the fundamental component of the library. Additionally, there could be other functional dependencies of other type class definitions that are not needed and are not yet discovered. Therefore, an investigation could be carried out to eliminate any possible unnecessary functional dependencies.

Applying the Same Approach in Haskell

A similar approach could be applied to Haskell, which is also a functional language that has a very similar syntax and type system as Purescript, to embed multiparty session type in Haskell programs. Although Haskell is a less popular option in developing web applications, it is capable of building both front-end and back-end applications. Unlike Purescript, Haskell lacks native row type support, posing a challenge for encoding the `Select` and `Branch` EFSM transitions. However, a recently new package for supporting row types in Haskell [26] could be an alternative to work around it.

Solution to Runing Multiple Session Runtimes that Handle UI Updates

If there are multiple session runtimes running concurrently in a front-end application, any update to a UI component of a particular session re-renders every UI component in the application. This is an issue discovered via implementing a chat application using the library. The cause of the problem is still unknown and a solution is needed to tackle this problem, as it breaks the application in this scenario.

Chapter 9

Appendices

9.1 WebRTC Purescript Bindings

9.1.1 Javascript Foreign Module Declarations

```
// module WebRTC.RTC
//
exports.hasRTC = typeof RTCPeerConnection === "function"

exports.newRTCPeerConnection = function(psConfig) {
    return function() {
        function nativeIceServer(psIceServer) {
            return psIceServer.value0;
        };
        return new RTCPeerConnection({
            bundlePolicy: psConfig.bundlePolicy.value0,
            iceServers: psConfig.iceServers.map(nativeIceServer),
            iceCandidatePoolSize: psConfig.iceCandidatePoolSize,
            iceTransportPolicy: psConfig.iceTransportPolicy.value0,
            peerIdentity: psConfig.peerIdentity.value0
        });
    };
};


exports.closeRTCPeerConnection = function(pc) {
    return function() {
        pc.close();
    };
};


exports.onclose = function(handler) {
    return function(pc) {
        return function() {
```

```
            pc.oniceconnectionstatechange = function() {
                if(pc.iceConnectionState == 'disconnected' || pc.
                    iceConnectionState == 'closed') {
                    handler();
                }
            }
        };
    };
};


exports.addStream = function(stream) {
    return function(pc) {
        return function() {
            pc.addStream(stream);
        };
    };
};


exports._signalingState = function(pc) {
    return function() {
        return pc.signalingState;
    };
};


exports.onicecandidate = function(f) {
    return function(pc) {
        return function() {
            pc.onicecandidate = function(event) {
                f(event)();
            };
        };
    };
};

exports.onconnectionstatechange = function(f) {
    return function(pc) {
        return function() {
            pc.onconnectionstatechange = function(event) {
                f(pc.connectionState)();
            };
        };
    };
};

exports.ontrack = function(f) {
    return function(pc) {
        return function() {
            pc.ontrack = function(event) {
```

```
                f(event)();
            };
        };
    };
};


exports._createOffer = function(pc) {
    return function(error, success) {
    pc.createOffer()
        .then(function(offer) {
            success(new RTCSessionDescription(offer))();
        })
        .catch(error);
        return function(a, b, cancelerSuccess) { cancellerSuccess(); };
    };
};


exports._createAnswer = function(pc) {
    return function(error, success) {
        pc.createAnswer(success, error);
        return function(a, b, cancelerSuccess) { cancellerSuccess(); };
    };
};


exports._setLocalDescription = function(desc) { return function(pc) {
    return function(error, success) {
        pc.setLocalDescription(desc, success, error);
        return function(a, b, cancelerSuccess) { cancellerSuccess(); };
    };
};};


exports._setRemoteDescription = function(desc) { return function(pc) {
    return function(error, success) {
        pc.setRemoteDescription(desc, success, error);
        return function(a, b, cancelerSuccess) { cancellerSuccess(); };
    };
};};


exports._iceEventCandidate = function(nothing) {
    return function(just) {
        return function(e) {
            return e.candidate ? just(e.candidate) : nothing;
        };
    };
};
```

```
exports.addIceCandidate = function(c) {
    return function(pc) {
        return function() {
            pc.addIceCandidate(new RTCIceCandidate(c));
        };
    };
};


exports.newRTCSessionDescription = function(s) {
    return new RTCSessionDescription(s);
};


exports.createDataChannel = function(s) {
    return function(pc) {
        return function() {
            var dc = pc.createDataChannel(s);
            return dc;
        };
    };
};


exports.send = function(s) {
    return function(dc) {
        return function() {
            if (dc.readyState != "open") return;
            dc.send(s);
        };
    };
};


exports.onmessage = function(f) {
    return function(dc) {
        return function() {
            dc.onmessage = function(m) {
                f(m.data)();
            };
        };
    };
};


exports.ondatachannel = function(handler) {
    return function(pc) {
        return function() {
            pc.ondatachannel = function(ev) {
                handler(ev.channel)();
```

```
            };
        };
    };
}


exports.onopen = function(handler) {
    return function(dataChannel) {
        return function() {
            dataChannel.onopen = handler;
        };
    };
}

exports.ondatachannelclose = function(handler) {
    return function(dataChannel) {
        return function() {
            dataChannel.onclose = handler;
        };
    };
}
```

9.1.2 Purescript Foreign Functions Declarations

```
module WebRTC.RTC

import ...

foreign import hasRTC :: Boolean

foreign import data RTCPeerConnection :: Type

data RTCIceServer
  = STUNServer { urls :: Array String }
  | TURNServer { urls :: Array String
               , username :: String
               , credential :: String
               }

data RTCSignalingState
  = Stable
  | HaveLocalOffer
  | HaveRemoteOffer
  | HaveLocalProvisionalAnswer
  | HaveRemoteProvisionalAnswer
  | UnknownValue String

derive instance eqRTCSignalingState :: Eq RTCSignalingState
```

```
type RTCConfiguration =
  { bundlePolicy :: Maybe String
  -- certificates :: (not yet implemented)
  , iceServers :: Array RTCIceServer
  , iceCandidatePoolSize :: Int
  , iceTransportPolicy :: String
  , peerIdentity :: Maybe String
  -- , rtcpMuxPolicy (at risk due to lack of implementor interest)
  }


defaultRTCConfiguration :: RTCConfiguration
defaultRTCConfiguration =
  { bundlePolicy: Nothing
  , iceServers: []
  , iceCandidatePoolSize: 5
  , iceTransportPolicy: "all"
  , peerIdentity: Nothing
  }


foreign import newRTCPeerConnection
  :: RTCConfiguration -> Effect RTCPeerConnection


foreign import closeRTCPeerConnection
  :: RTCPeerConnection -> Effect Unit


foreign import onclose
  :: Effect Unit -> RTCPeerConnection -> Effect Unit


foreign import addStream
  :: MediaStream -> RTCPeerConnection -> Effect Unit


-- Getting the signalling state
foreign import _signalingState :: RTCPeerConnection -> Effect String


stringToRTCSignalingState :: String -> RTCSignalingState
stringToRTCSignalingState =
  case _ of
    "stable"               -> Stable
    "have-local-offer"     -> HaveLocalOffer
    "have-remote-offer"    -> HaveRemoteOffer
    "have-local-pranswer"  -> HaveLocalProvisionalAnswer
    "have-remote-pranswer" -> HaveRemoteProvisionalAnswer
    other                  -> UnknownValue other
```

```
signalingState :: RTCPeerConnection -> Effect RTCSignalingState
signalingState pc = stringToRTCSignalingState <$> _signalingState pc


foreign import data IceEvent :: Type


type RTCIceCandidate = { sdpMLineIndex :: Int
                       , sdpMid :: String
                       , candidate :: String
                       }


foreign import _iceEventCandidate
  ::     a. Maybe a ->
              (a -> Maybe a) ->
              IceEvent ->
              Maybe RTCIceCandidate


iceEventCandidate :: IceEvent -> Maybe RTCIceCandidate
iceEventCandidate = _iceEventCandidate Nothing Just


foreign import addIceCandidate
  :: RTCIceCandidate -> RTCPeerConnection -> Effect Unit


foreign import onicecandidate
  :: (IceEvent -> Effect Unit) -> RTCPeerConnection -> Effect Unit

foreign import onconnectionstatechange
  :: (String -> Effect Unit) -> RTCPeerConnection -> Effect Unit


type RTCTrackEvent = { streams :: Array MediaStream, track :: MediaStreamTrack }


foreign import ontrack
  :: (RTCTrackEvent -> Effect Unit) -> RTCPeerConnection -> Effect Unit


-- foreign import data RTCSessionDescription :: Type


foreign import newRTCSessionDescription
  :: { sdp :: String, "type" :: String } -> RTCSessionDescription
```

```
foreign import _createOffer
  :: RTCPeerConnection -> EffectFnAff RTCSessionDescription


createOffer :: RTCPeerConnection -> Aff RTCSessionDescription
createOffer = fromEffectFnAff <<< _createOffer


foreign import _createAnswer
  :: RTCPeerConnection -> EffectFnAff RTCSessionDescription


createAnswer :: RTCPeerConnection -> Aff RTCSessionDescription
createAnswer = fromEffectFnAff <<< _createAnswer


foreign import _setLocalDescription
  :: RTCSessionDescription -> RTCPeerConnection -> EffectFnAff Unit


setLocalDescription :: RTCSessionDescription -> RTCPeerConnection -> Aff Unit
setLocalDescription desc pc = fromEffectFnAff $ _setLocalDescription desc pc


foreign import _setRemoteDescription
  :: RTCSessionDescription -> RTCPeerConnection -> EffectFnAff Unit


setRemoteDescription :: RTCSessionDescription -> RTCPeerConnection -> Aff Unit
setRemoteDescription desc pc = fromEffectFnAff $ _setRemoteDescription desc pc


foreign import data RTCDataChannel :: Type

foreign import createDataChannel
  :: String -> RTCPeerConnection -> Effect RTCDataChannel

foreign import send
  :: String -> RTCDataChannel -> Effect Unit

foreign import onmessage
  :: (String -> Effect Unit) -> RTCDataChannel -> Effect Unit

foreign import ondatachannel
  :: (RTCDataChannel -> Effect Unit) -> RTCPeerConnection -> Effect Unit

foreign import onopen :: (Effect Unit) -> RTCDataChannel -> Effect Unit

foreign import ondatachannelclose :: (Effect Unit) -> RTCDataChannel -> Effect Unit

data ServerType
```

```
  = STUN { urls :: NonEmpty Array String }
  | TURN { urls :: NonEmpty Array String, credentialType :: Maybe String, credential
      :: Maybe String, username :: Maybe String }

instance serverTypeEncodeJson :: EncodeJson ServerType where
  encodeJson (STUN s) = jsonSingletonObject "urls" (encodeJson s.urls)
  encodeJson (TURN t) = (
    "urls" := t.urls
    ~> "credentialType" := t.credentialType
    ~> "credential" := t.credential
    ~> "username" := t.username
    ~> jsonEmptyObject
  )

instance serverTypeDecodeJson :: DecodeJson ServerType where
  decodeJson json' = getTurn json' <|> getStun json'
    where
      getTurn json = do
        obj <- decodeJson json
        credentialType <- getField obj "credentialType"
        credential <- getField obj "credential"
        username <- getField obj "username"
        urls <- getField obj "urls"
        pure $ TURN { credentialType, credential, username, urls }
      getStun json = do
        obj <- decodeJson json
        urls <- getField obj "urls"
        pure $ STUN { urls }

newtype RTCSessionDescription = RTCSessionDescription { sdp :: String, "type" ::
    String }

rtcSessionDescriptionSdp :: RTCSessionDescription -> String
rtcSessionDescriptionSdp (RTCSessionDescription r) = r.sdp

rtcSessionDescriptionType :: RTCSessionDescription -> String
rtcSessionDescriptionType (RTCSessionDescription {"type" : t}) = t

instance rTCSessionDescriptionEncodeJSON :: EncodeJson RTCSessionDescription where
  encodeJson (RTCSessionDescription {"sdp" : sdp, "type" : t}) =
    ("sdp" := sdp
    ~> "type" := t
    ~> jsonEmptyObject)

instance rTCSessionDescriptionDecodeJSON :: DecodeJson RTCSessionDescription where
  decodeJson json = do
    obj <- decodeJson json
    sdp <- getField obj "sdp"
    t <- getField obj "type"
    pure $ RTCSessionDescription { "sdp" : sdp, "type" : t }
```

Bibliography

[1] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, & Nicholas Ng: The scribble protocol language. In *International Symposium on Trustworthy Global Computing* (2013), Springer, pp. 22–41. pages 8

[2] Raymond Hu & Nobuko Yoshida (2016): Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering*, LNCS 9633, Springer, pp. 401–418. pages 5, 14

[3] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *31st European Conference on Object525 Oriented Programming*, LIPIcs 74, Schloss Dagstuhl, pp. 24:1–24:31. pages 15

[4] Rumyana Neykova, Raymond Hu, Nobuko Yoshida & Fahd Abdeljallal (2018): A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. In *27th International Conference on Compiler Construction*, ACM, pp. 128–138. pages 15

[5] Nicholas Ng, Jose G.F. Coutinho & Nobuko Yoshida (2015): Protocols by Default: Safe MPI Code Generation based on Session Types. In *24th International Conference on Compiler Construction*, LNCS 9031, Springer, pp. 212–232. pages 15

[6] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng & Nobuko Yoshida (2019): Distributed Programming Using Role Parametric Session Types in Go. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, ACM, pp. 1–30. pages 15

[7] Robin Milner (1978): A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, 17(3), pp. 348–375. pages 4

[8] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou & Nobuko Yoshida (2014): Structuring Communication with Session Types. In *Lecture Notes in Computer Science book series*, LNCS, pp. 105–127. pages 4

[9] Kohei Honda, Nobuko Yoshida, & CARBONE, Marco (2016): Multiparty Asynchronous Session Types. In *Journal of the ACM*, pp. 1–67. pages 4, 5

[10] Rumyana Neykova (2013): Session Types Go Dynamic or How to Verify Your Python Conversations. In *5th International Workshop on Programming Language Approaches to Concurrency and Communication-centric Software*, vol. 137 of EPTCS, Open Publishing Association, pp. 95–102. pages 5

[11] Nicholas Ng, Nobuko Yoshida, & Kohei Honda (2012): Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *50th International Conference on Objects, Models, Components, Patterns*, vol. 7304 of LNCS, Springer, pp. 202–218. pages 5

[12] Nobuko Yoshida: Lecture notes for CO406: Introduction to Session Types. pages 5

[13] Nobuko Yoshida, & Lorenzo Gheri (2019): A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology Lecture Notes in Computer Science*, LNCS, Springer, pp. 73–93. pages 5

[14] Jonathan King, Nicholas Ng & Nobuko Yoshida (2019): Multiparty Session Type-safe Web Development with Static Linearity. In *Electronic Proceedings in Theoretical Computer Science*, pp. 35–46. pages 2, 9, 12, 17, 74

[15] Raymond Hu, Kohei Honda, Rumyana Neykova & Nobuko Yoshida (2014): Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python. In *Formal Methods in System Design*, volume 8174 of LNCS, Springer, pp. 197-225. pages 10

[16] Simon Fowler (2019): Model-View-Update-Communicate Session Types meet the Elm Architecture, arXiv:1910.11108 [cs.PL]. pages 13

[17] Keigo Imai, Nobuko Yoshida & Shoji Yuen (2017): Session-ocaml: a session-based library with polarities and lenses. In *19th International Conference on Coordination Models and Languages*, LNCS 10319, Springer, pp. 99–118 pages 13

[18] Alexey Melnikov & Ian Fette (2011): The WebSocket Protocol. RFC 6455. Available at https://rfc-editor.org/rfc/rfc6455.txt. pages 1, 32

[19] Justin Uberti and Peter Thatcher (2011): *WebRTC*. `https://webrtc.org`. pages 1, 32, 33

[20] MDN Contributors (2019): *RTCPeerConnection createOffer API*. `https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/createOffer`. pages 37

[21] MDN Contributors (2019): *RTCPeerConnection createAnswer API*. `https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/createAnswer`. pages 37

[22] M. Perkins & et al. (2006): *SDP: Session Description Protocol*. `https://tools.ietf.org/html/rfc4566`. pages 34

[23] MDN Contributors (2020): *RTCPeerConnection setLocalDescription API*. https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/setLocalDescription. pages 39

[24] MDN Contributors (2019): *Using WebRTC data channels*. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Using_data_channels. pages 41

[25] The node-webrtc contributors (2020): *node-webrtc*. https://github.com/node-webrtc/node-webrtc. pages 75

[26] Daniel Winograd-Cort & Matthew Farkas-Dyck (2020): *row-types:Open Records and Variants*. https://hackage.haskell.org/package/row-types. pages 75

[27] MDN Contributors (2020): *MediaDevices.getUserMedia API*. https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia pages

[28] The Purescript contributors (2013): *Purescript*. https://www.purescript.org/. pages 2, 60

[29] Gary Burgess & Nathan Faubion (2018): *Purescript asynchronous variables*. https://pursuit.purescript.org/packages/purescript-avar/3.0.0/docs/Effect.Aff.AVar. pages 46

[30] Jonathan King (2019): *MEng Individual Project Final Report*. pages 61, 63

[31] Jonathan King (2019): *Purescript-scribble*. https://github.com/jonathanlking/purescript-scribble. pages vi, 60, 74

[32] MDN Contributors (2020): *RTCPeerConnection setRemoteDescription API*. https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/setRemoteDescription. pages 39

[33] The spago contributors (2020): Spago. https://github.com/purescript/spago.

pages 60