A dynamic approach to protocol conformance verification using multiparty session types

Bachelor's Project Computing Science

July 2022

Student: R.A. Dobre (s4051203)

First supervisor: B. van den Heuvel, MSc.

Second supervisor: J.A. Pérez, Prof. Dr.

Abstract

In distributed systems, communication is realised through the interchange of messages between various participants. Session types abstract this communication by specifying the types of messages exchanged between two peers. The theory of session types addresses the correctness of such *message-passing* programs by analysing their source code against communication protocols. We propose to implement a tool based on the theory of session types by Van den Heuvel and Pérez in 2022, to verify the correctness of message-passing programs in a distributed and asynchronous manner.

We implement a tool to verify each component of a message-passing system locally. To this end, the tool will create new, distributed components to act on behalf of each communicating party, inspect and verify the message exchanges, and assess the conformance to a protocol using multiparty session types. We then test the aforementioned tool against an authorization protocol. The implemented tool provides a new, dynamic method to assess the conformance to a protocol of multiple participants.

Contents

1 Introduction

With the ever-increasing scale of software, ensuring the correct functionality of a program is often difficult. The spotlight has shifted towards distributed systems, relying on networks and underlying protocols to steer communication. Applications frequently utilise the services of other applications for various reasons, such as to avoid reinventing the wheel, faster time to market and more. The dependencies on other applications come with several implications, including the need to use a potentially poorly documented interface. Additionally, the given interfaces are often prone to unforeseen changes.

Testing is a popular approach to verifying a system's correct functionality, as described in [25]. Nevertheless, testing can never encompass all possible scenarios. Thus, other methods, such as formal program correctness theory relying on logic and mathematics, have been developed. Session types represent suitable candidates for verifying the conformance of a message-passing program to a given protocol, as they abstract the communication by means of message-passing [24].

The current state of the art focuses on a static approach to protocol conformance verification through the analysis of the source code of the communicating parties [24]. This paper proposes a new, dynamic approach to assessing protocol conformance by inspecting and verifying the messages exchanged at runtime. We plan to expand the availability of tools relying on session types, such as [1, 2, 7, 10, 14]. Subsequently, this gives rise to the following research question, which will be the backbone of our research project: *How can we adapt the theory of static protocol conformance verification to an application of dynamic verification?* Intuitively, implementing a tool relying on dynamic verification methods introduces multiple research questions regarding the tool's design. The most prominent design challenges of the research project consist of modelling the interface for the protocol's specification and finding a suitable data structure to facilitate efficient verification of the observed messages. The research problem has strong roots in the Program Correctness field of Computing Science, and it is tightly connected to logic and process calculi.

Our research project will extend the current state of the art by implementing a library to dynamically assess protocol conformance, together with a test suite comprised of an authorization protocol and a weather protocol. Our library will observe the messages exchanged during the execution of the protocol, and it will use them to determine whether the participants adhere to the protocol. To this end, it will create new components for each participant, routers, meant to act as middleware between a protocol participant and the rest of the network. Consequently, the protocol participants will exchange messages with other participants through their own routers. In turn, routers will listen to incoming messages, they will verify them internally, and then they will forward them to their correct recipients. Figure 1 illustrates a network of participants before and after our tool is deployed.

The research project will be of great use to the industry, as the tool provides a means to enforce a protocol in a distributed system with minimal changes to the implementations of the communicating participants. Consequently, we find that our research project will have a significant impact on the research community revolving around multiparty session types, as we plan to further the presence of multiparty session types in the realm of dynamic verification methods. Furthermore, as far as we are aware, this is the first dynamic verification tool for multiparty session types written in JavaScript that does not rely on Scribble [8], a protocol specification language.

Section 2 describes the preliminaries needed for understanding the library. It discusses routers, their theoretical foundations, and the properties that they ensure. Multiparty session types and the theory underlying routers as defined by Van den Heuvel and Pérez [23] are then outlined, followed by a description of finite state machines. Section 3 details the objectives of the library, its requirements and describes its implementation. Section 4 describes the implementation of an example authorization protocol outlined in [23] and the results obtained by testing the library against it. Section 5 describes the usability of the tool in the scenario where some participants may not be compatible with the tool. Section 6 details the implementation of a transpiler meant to ease the process of specifying the protocol, and Section 7 describes related and future work. Finally, Section 8 concludes the paper, and then acknowledges all the people that had significant contributions to the project.

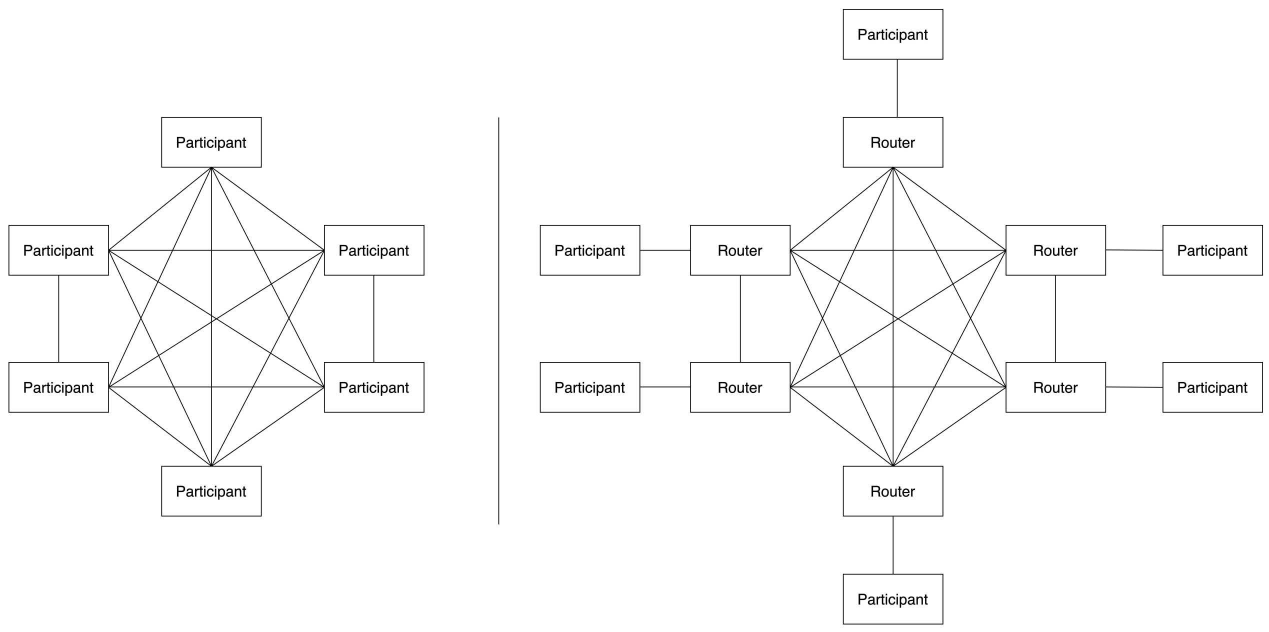


Figure 1: Network of participants vs network of routers and their corresponding participants.

2 Preliminaries

This section describes the preliminaries needed to understand the underlying theory of the implemented library. It first introduces routers as the central component in the given library, together with their purpose and objectives. Thereafter, this section recapitulates multiparty session types to demonstrate how protocols can be abstracted and verified in a network of communicating participants. Next, this section dives into relative types, a framework for multiparty session types developed by Van den Heuvel and Pérez [23], and its purposes in the current library. At last, this section describes finite state machines and how they can be used to encode the behaviour of routers.

2.1 Routers

One of the fundamental requirements of the implemented library is verifying protocol conformance with minimal changes to the implementations of the communicating participants. However, to satisfy the aforementioned requirement, there is a need to observe and verify every message exchange that occurs in the given system. Given the messages sent between the communication participants, one can easily compare them against a protocol to see whether the participants respect it. However, the question arises, how can one efficiently observe the messages exchanged in a system?

The library that we implement uses new components to encapsulate each communication participant. Thus, every message will go through these components in order to reach its correct recipient, enabling them to oversee and verify every exchange. Consequently, this results in a dynamic adaptation of a static verification method [23].

Before the creation of modern telephone systems, people had to call a telephone switchboard centre first to connect to the actual receiver. Therefore, the switchboard centre would be in charge of organising and managing the communication between every pair of participants. *Centralised* verification methods rely on the same principle: have a central component acting as a middleman for every interaction to inspect and verify every message.

On the opposite side of centralised verification methods lie the *decentralised* methods. Instead of calling a switchboard centre shared between all participants, each participant will now have its own switchboard. Thus, each participant switchboard will be in charge of organising and managing the calls to other participants. As

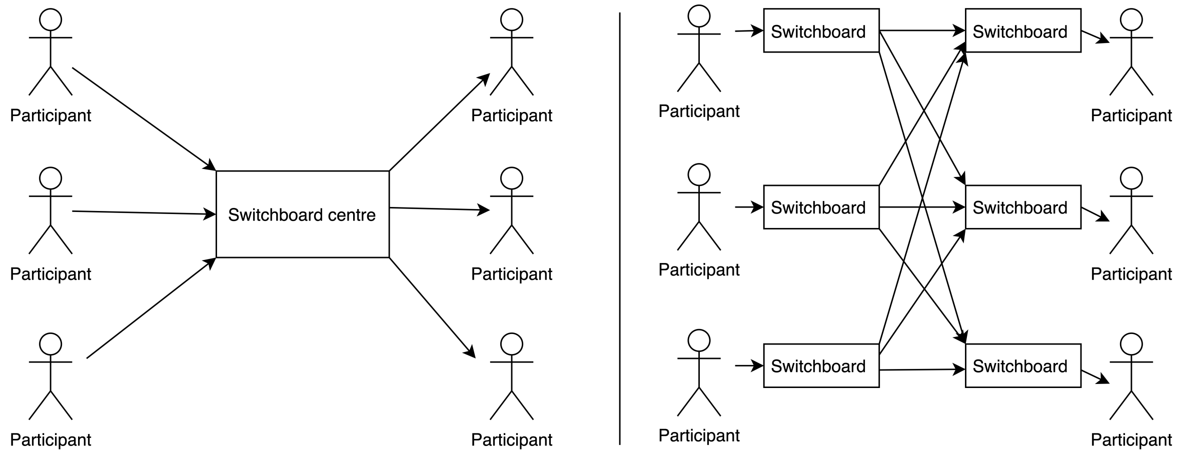


Figure 2: Centralised and decentralised switchboards in telephone systems.

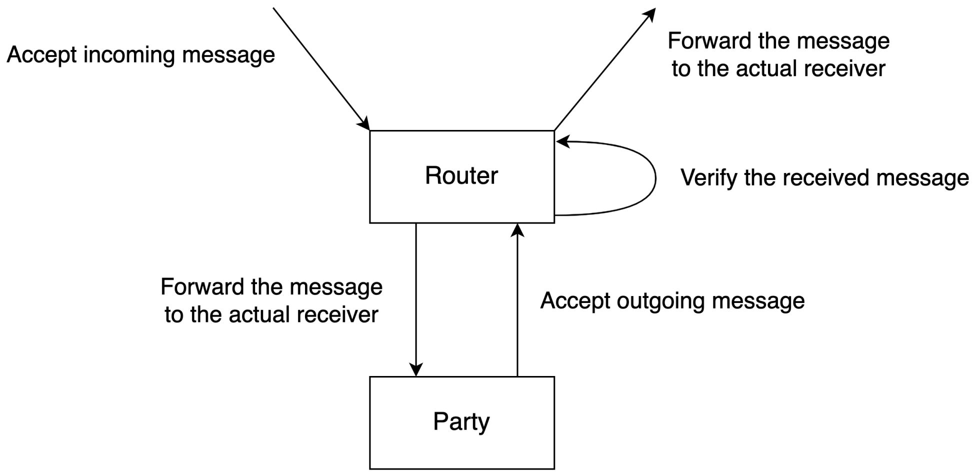

Accept incoming message
Forward the message to the actual receiver
Router
Verify the received message
Forward the message to the actual receiver
Accept outgoing message
Party

Figure 3: Overview of the actions taken by a router upon the receipt of a message.

such, the responsibility of the central switchboard will be distributed to the switchboard of every participant, resulting in a decentralised approach. Decentralised verification methods use multiple components distributed at each participant to verify the conformance to the given protocol. The antithesis between centralised and decentralised methods can be seen in Figure 2 using the example of switchboard systems.

Both centralised and decentralised methods offer several advantages and disadvantages. While centralised methods might be easier to set up initially and are conceptually more straightforward, they suffer from scalability issues. For instance, a centralised method might work perfectly for a switchboard centre connecting just six participants. However, it quickly becomes infeasible if another town of six hundred participants needs to connect to the same switchboard. On the other hand, decentralised methods can easily tackle increasing scalability, but they are more complicated to configure and use.

The tool we implement uses a decentralised approach to protocol conformance verification. To this end, the tool will generate new components, routers. For each participant, the library will create a router, and it will serve as a middleman between the participant and the rest of the system. The fundamental role of the router is to accept messages from the rest of the network or the participant, verify them internally to see whether they are in accordance with the given protocol, and then forward them to the correct recipient. Figure 3 shows an overview of the steps taken by a router upon receiving a message.

Routers will oversee and verify the message exchanges. To this end, routers will require access to the types of the exchanged messages and their corresponding order. However, the description of the system-wide message exchanges will not suffice, because the routers can only observe the messages that involve their corresponding participant. Hence, routers will employ relative types, which we further describe in Section 2.3. Relative types represent a framework of multiparty session types, which we introduce in Section 2.2.

2.2 Multiparty Session Types

Protocols represent sequences of actions meant to achieve a goal. For example, consider the communication between two people who want to become acquaintances. The individuals usually introduce themselves by exchanging their name, age, and occupation. After the general introductions, the individuals then communicate about whatever they desire. The structure in the initial communication between the two allows us to introduce a protocol for meeting strangers. In the aforementioned example, the protocol for getting acquainted with a stranger is denoted by exchanging an individual's name, age, and occupation.

Protocols in textual form are helpful for thoroughly understanding the communication and the context between the different participants. Nonetheless, a way is needed to reason about them from a mathematical and logical standpoint. The textual form does not suffice, as it contains too many, possibly unnecessary, details and ambiguities. For example, the aforementioned protocol does not impose a strict ordering on the messages the participants exchange, nor does it specify how they communicate each personal attribute. There is a recurring need to abstract away these details and ambiguities.

Multiparty session types abstract the communication between two or more participants by explicitly specifying what messages are exchanged between which participants, their types, and the order in which they are exchanged. Multiparty session types are a generalisation of binary session types, which denote the sequence of exchanged messages between two participants.

We can rewrite the aforementioned protocol through the perspective of multiparty session types but in textual form. We need to state the messages exchanged between the two strangers, their types, and their corresponding order. The protocol starts with the two individuals exchanging names. We can abstract this interaction by specifying that one individual sends their name as a string, accompanied by a label such as *name*, to the other participant. Thereafter, the second individual proceeds to do the same, and the participants move on to the next stage, where they will exchange their age.

The specification of the interactions from a vantage point is represented through global types. As the name suggests, they cover the system-wide message exchanges. On the other hand, local types express the contributions of a single participant to the given protocol. They are obtained by projecting the global type onto a participant. They are usually employed in static verification methods, through the analysis of the implementation of a specific participant, or in dynamic verification methods, by inspecting and verifying the messages exchanged at runtime.

For the specification of local and global types, we will be using the syntax defined in the works of Yoshida and Gheri [24]. Global types (G) can be specified by using:

- The type $a \rightarrow b : \{i(T_i)\,.\,G_i\}_{i \in I}$ denotes that a chooses and sends a message to b with a label i appearing in the set I and a value of type T_i, after which the protocol continues as G_i. If the exchange consists of a single label, we omit the curly brackets. Furthermore, we omit unit types, denoting that no value will be sent.
- The type $\mu X.G$ declares recursion on the variable X.
- The type X denotes a call to the recursive type defined on X.
- The type *end* defines the end of the protocol.

We can write the protocol for meeting new people defined previously using the following global type. The protocol is simple, as it does not contain any recursion or choices. Nevertheless, it provides insight into the syntax of multiparty session types and how they are used.

$$\begin{aligned} G_{\mathsf{meet}} = {} & a \rightarrow b : \mathsf{name}(str)\,.\, b \rightarrow a : \mathsf{name}(str)\,.\, a \rightarrow b : \mathsf{age}(int)\,.\, b \rightarrow a : \mathsf{age}(int) \\ & .\, a \rightarrow b : \mathsf{occupation}(str)\,.\, b \rightarrow a : \mathsf{occupation}(str)\,.\, end \end{aligned} \tag{1}$$

The global type G_{meet} denotes the protocol for two people getting acquainted. Two participants, a and b, are involved in the given protocol. It starts with one participant a, sending to the other participant b

a message labelled name, and its name as a string. The other participant replies with a similar message. Then, a proceeds to send to b a message labelled age, and its age as an integer. Next, b replies with a similar message. Thereafter, a sends to b a message labelled $\mathsf{occupation}$, together with its occupation given as a string. Subsequently, b replies with a message labelled $\mathsf{occupation}$ and its occupation as a string.

In the rest of the paper, we are using the Client-Server-Authorization protocol by Van den Heuvel and Pérez [23], adapted from [22]. It involves three participants, a server (s), a client (c), and an authorization service (a). The protocol denotes an authorization scenario when a specific service asks a user to authorize itself or quit. As the name suggests, the goal of this protocol is authorization with the use of another service. The global type of the Client-Server-Authorization protocol is as follows:

$$G_{\mathsf{auth}} = \mu X \,.\, s \to c : \{\mathsf{login} \,.\, c \to a : \mathsf{passwd}(str) \,.\, a \to s : \mathsf{auth}(bool) \,.\, X, \quad \mathsf{quit} \,.\, c \to a : \mathsf{quit} \,.\, end\} \tag{2}$$

The global type G_{auth} starts by defining recursion on the variable X. Thereafter, the server sends to the client a message to login or quit. If the server sends the login message to the client, the client then sends a message labelled passwd and a password as a string to the authorization server. Next, the authorization service sends a message labelled auth to the server, together with a boolean value, to indicate whether the user is authorized or not. Then, it loops back to the beginning of the protocol due to the recursive call. If the server sends the quit message to the client, the client sends a quit message to the authorization service, and the protocol terminates.

2.3 Relative Types and Relative Projection

In their work, Van den Heuvel and Pérez introduce a new framework based on multiparty session types [23]. Their framework uses a decentralised approach for protocol conformance verification and offers several benefits, such as the guarantee that participants will not have to wait indefinitely for other participants. As such, protocol conformance will be verified in a distributed manner, local to each participant. Furthermore, routers will be generated for each participant, and they will encapsulate their corresponding participant, intervening in every message exchange with it. Thus, the original network of participant implementations will become a network of routers and their corresponding participant implementations.

Every router will be connected to the router of every other participant. Thus, the local type resulting from the projection of the global type onto a participant will no longer suffice. Van den Heuvel and Pérez introduce the notion of relative types, which represent the communication between pairs of different participants. Routers will employ relative types to express what messages must be exchanged with another router or participant.

Binary session types abstract the exchanges between two participants, whereas relative types express the interactions between a pair of participants. Thus, the question arises, how are the two concepts different? Suppose that an individual is in a restaurant and wishes to order food. The individual must decide between two meals, meal_1 and meal_2. If the individual settles on the first meal, it tells the waiter the preferred option, and then the waiter tells the chef to start the preparation. If the individual settles on the second meal, the preferred option is communicated to the waiter, but this time the waiter tells the chef to order some items from a store. Binary session types would be unable to capture the entire communication between the individual, the waiter and the chef since they are restricted to two participants. However, they might be able to represent the communication between the individual and the waiter, and the communication between the waiter and the chef. Nevertheless, the initial exchange between the individual and the waiter impacts the communication between the waiter and the chef. Binary session types cannot express this influence of the communication between the waiter and the individual on the communication between the waiter and the chef. Relative types are able to capture this type of influence between the participants of a protocol.

Van den Heuvel and Pérez define the choices involving a third participant that affect the communication between the current pair of participants as non-local choices. Non-local choices will prove to be vital in the architecture of our routers, as they will enable them to coordinate on the choices of other participants.

Referring to the aforementioned protocol for ordering food, the initial exchange between the individual and the waiter represents a non-local choice for the communication between the waiter and the chef. Thus, a way is needed to inform the chef of whatever the individual chose initially, so that the chef knows what message to expect, and what to do next. Van den Heuvel and Pérez introduce *dependencies* to tackle this issue. Dependencies capture the need to forward particular choices to other participants, such that they would be able to coordinate with the rest of the participants. In the example mentioned above, the dependency would indicate that the waiter must forward the choice of the individual to the chef such that it can coordinate with the waiter.

We will use the syntax for relative types (R) defined in [23]. In case of an exchange between the current pair of participants, it suffices to specify just the sender, as it is known in advance who the recipient is. The syntax for the recursive call, definition and the end of the protocol is shared with the syntax of global types. In the syntax given below, the participants involved in the current relative type are a and b.

- The type $a\{i(T_i) \,.\, R_i\}_{i\in I}$ expresses an internal choice at a, which will then send a message labelled i and a value of type T_i to b, and then continue as R_i. Similar to global types, if there is a single choice, we omit the curly brackets, and we do not specify unit types.

- The type $a?c\{i \,.\, R_i\}_{i\in I}$ denotes a dependency on a third participant c, different from the current pair (a, b). The dependency implies a direct influence of a message between a and c on the communication between a and b. In order to coordinate with b on this choice, a must forward to b the label received from c.

- The type $a!c\{i \,.\, R_i\}_{i\in I}$ denotes a dependency on a third participant c, different from the current pair (a, b). The dependency implies a direct influence of a message between a and c on the communication between a and b. In order to coordinate with b on this choice, a must forward to b the label that a sent to c.

Relative types are computed by projecting the global type onto each pair of participants. To this end, Van den Heuvel and Pérez introduce the notion of *relative projection*. Throughout the rest of this section, we will use the rules of relative projection and its definition expressed in [23].

To illustrate relative types and relative projection, let us consider an example first. We will take the aforementioned protocol for ordering food in a restaurant, and we will modify it such that:

- the manager (m) will ask the waiter (w) first to go and take the order of the individual (i);

- after ordering a meal, the individual will continually order more.

We can express the updated protocol as $G_{\mathsf{restaurant}}$.

$$\begin{aligned} G_{\mathsf{restaurant}} = m \to w : \mathsf{takeOrder} \,.\, \mu X \,.\, i \to w : \{&\mathsf{meal}_1 \,.\, w \to c : \mathsf{prepareFood}(str) \,.\, X, \\ &\mathsf{meal}_2 \,.\, w \to c : \mathsf{orderItems}(str) \,.\, X \;\} \end{aligned} \tag{3}$$

From the global type, we can observe that the manager interacts initially with the waiter, and then the individual orders infinitely many meals. This is possible through the recursive definition before the exchange between the individual and the waiter, and through the recursive calls at the end of each branch. However, considering the global type, the manager is only involved in the initial exchange and has no contributions afterwards. Additionally, the manager is not involved in further interactions with the individual or the waiter. One could argue that the manager should not sit around and observe in a restaurant. Since the manager's contribution to the protocol comes down to a single message exchange, the relative types involving the manager should reflect this: the protocol for the manager should end after the first message exchange.

Before diving into the formal relative projection rules, we inspect $G_{\mathsf{restaurant}}$ and see what cases come up when computing the relative projection for a pair of participants, to get an intuition of what will follow. Take the relative projection for the pair (m, w): looking at the first exchange, both participants are involved in the given interaction. Thus, the relative projection should express that m sends a message to w with the label $\mathsf{takeOrder}$. The global type continues with a recursive definition on the variable X. The manager is not involved in the rest of the protocol, and it has no influence whatsoever on its continuation. Hence, the relative projection should yield the end of the protocol. Taking the pair (i, w), only w is involved in the first exchange with the manager. Nevertheless, the exchange only consists of a single branch, so it does not impact the communication between the individual and the waiter. Thus, they can carry on with the continuation of the protocol. The continuation of the protocol is a recursive definition on the variable X. The individual and the waiter interact after the recursive definition, as the individual can order as many dishes as desired. Hence, the relative projection should yield a recursive definition since the participants have further interactions. The following exchange involves the pair, so the relative projection should express that the individual sends the waiter a message labelled either meal_1 or meal_2. In the continuations of the branches, the exchanges do not involve both participants of the current pair, nor do they contain multiple branches. Thus, they can move forward with the continuation of the protocol, encountering a recursive call on the variable X. Recursive calls projected onto a pair of participants will remain unchanged. The relative projection on the pair (c, w) is rather complex, because the exchange between the individual and the waiter represents a non-local choice for the communication between the cook and the waiter. Thus, the relative projection should capture this dependency and express it in terms of the obligation of the waiter to forward the message that it received from the individual to the chef.

The syntax of global types implies four different cases for which the projection needs to be computed: a direct exchange between two participants, a recursive declaration, a recursive call, and the end of the protocol. We will follow the syntax and the notation defined in [23]: the projection of a global type G on a pair of participants a and b is denoted $G \upharpoonright (a, b)$.

- An exchange projected onto a pair of participants $(x \to y : \{i(T_i) \,.\, G_i\}_{i \in I}) \upharpoonright (a, b)$ entails three possible scenarios:
  - If the pair involved in the exchange (x, y) corresponds exactly to the pair (a, b), the relative projection is easily computed, as we simply need to specify the sender of the message and then compute the projection for each branch: $a\{i(T_i) \,.\, (G_i \upharpoonright (a, b))\}_{i \in I}$;
  - If the pair involved in the exchange (x, y) corresponds to the pair (b, a), the relative projection is similar to the previous case: $b\{i(T_i) \,.\, (G_i \upharpoonright (a, b))\}_{i \in I}$;
  - If the interaction is not limited to the pair (a, b), an additional function *ddep* is needed to uncover any dependencies: $ddep((a, b), x \to y : \{i(T_i) \,.\, G_i\}_{i \in I})$. The function *ddep* takes as arguments the pair for which the projection is computed and the given exchange. We discuss the *ddep* function after the enumeration of the projection rules.
- A recursive declaration projected onto a pair of participants $(\mu X \,.\, G) \upharpoonright (a, b)$ entails two different cases:
  - If the projection of the continuation G onto the pair (a, b) contains any interactions, or ends in a recursive call on a different variable than X, then the continuation of the protocol is relevant for the pair (a, b). Formally speaking, Van den Heuvel and Pérez denote the aforementioned property as the *contractiveness* of the relative type on the given recursive variable. Thus, the recursive declaration remains the same, followed by the projection of the continuation of the protocol: $\mu X \,.\, (G \upharpoonright (a, b))$, given that the projection on the continuation is defined.
  - If any of the previously mentioned conditions do not hold (either it is not defined, or the projection of the continuation does not involve the participants a and b), then the projection of the recursive declaration yields an *end*.
- A recursive call projected onto a pair of participants $X \upharpoonright (a, b)$ is simply X;
- The end of the type projected onto a pair of participants $end \upharpoonright (a, b)$ is simply *end*.

The *ddep* function examines, given a pair of different participants and an exchange, the exchange and determines whether or not it impacts the communication between the pair. The function expects that at least one of the participants of the first pair is different from the participants involved in the exchange. Therefore, the function first checks if the exchange between the two participants is a non-local choice. To this end, the function compares the projections of all branches. If they are identical, then no matter what message is exchanged between the two participants, the communication between the pair will not be affected. Thus, the *ddep* function returns the projection of one of the equivalent branches. If the branches are not equal, the message exchange is a non-local choice, as the communication between the given pair will differ in some branches. If none of the participants in the given pair is involved in the exchange, then the projection is undefined, as the pair would have to coordinate based on the exchange between two different participants. Thus, the global type to be projected is not well-defined, because it cannot be projected onto all the pairs of participants. Alternatively, if the exchange includes one of the participants from the given pair, the function *ddep* returns a dependency involving that participant of the exchange. According to the previously discussed cases, the *ddep* function is as follows, where we assume that we are computing the projection for the pair (a, b).

$$ddep((a,b), x \to y : \{i(T_i) \ . \ G_i\}_{i \in I}) = \begin{cases} (G_k \upharpoonright (a,b)) & \text{if } \forall i,j \ (G_i \upharpoonright (a,b)) = (G_j \upharpoonright (a,b)) \\ & \text{for any } k \in I \\ a!y\{i \,.\, (G_i \upharpoonright (a,b))\}_{i \in I} & \text{if } a = x \\ a?x\{i \,.\, (G_i \upharpoonright (a,b))\}_{i \in I} & \text{if } a = y \\ b!y\{i \,.\, (G_i \upharpoonright (a,b))\}_{i \in I} & \text{if } b = x \\ b?x\{i \,.\, (G_i \upharpoonright (a,b))\}_{i \in I} & \text{if } b = y \end{cases} \tag{4}$$

It is helpful to consider an example to illustrate the relative projection and its workings. We can use the Client-Server-Authorization protocol, for which Equation (2) shows the global type. We are interested in the protocol between the Server and the Authorization service, so we will compute their relative type by projecting G_{auth} onto the pair (s, a). Consequently, we will apply the definition of relative projection inductively on the structure of G_{auth}.

The global type starts with a recursive declaration on the variable X. By definition, $(\mu X \,.\, G) \upharpoonright (s, a) = \mu X \,.\, (G \upharpoonright (s, a))$ if $(G \upharpoonright (s, a))$ is defined and contractive, and *end* otherwise. Thus, we need to compute the projection of the continuation of the global type and analyse it to see whether it is defined and contractive. The following interaction involves an exchange between the server and the client. Since the pair (s, a) does not correspond to either (s, c) or (c, s), we need to apply the function *ddep* with the arguments (s, a) and $s \to c : \{\mathsf{login} \,.\, G_{\text{login}}, \ \mathsf{quit} \,.\, G_{\text{quit}}\}$, where G_{login} represents the continuation of the login branch, and G_{quit} represents the continuation of the quit branch. The *ddep* function will check whether $G_{\text{login}} \upharpoonright (s, a) = G_{\text{quit}} \upharpoonright (s, a)$. Hence, it needs the relative projection of G_{login} onto (s, a) and the relative projection of G_{quit} onto (s, a).

$$G_{\text{login}} \upharpoonright (s, a) = (c \to a : \mathsf{passwd}(\mathit{str}) \,.\, a \to s : \mathsf{auth}(\mathit{bool}) \,.\, X) \upharpoonright (s, a) \tag{5}$$

The exchange between (c, a) does not involve the pair (s, a), so the *ddep* function will be called, with (s, a) as the first argument, and $c \to a : \mathsf{passwd}(\mathit{str}) \,.\, a \to s : \mathsf{auth}(\mathit{bool}) \,.\, X$ as the second argument. Since the exchange has only one branch, the *ddep* function will return $(a \to s : \mathsf{auth}(\mathit{bool}) \,.\, X) \upharpoonright (s, a)$. The current interaction involves both the server and the authorization service, so the corresponding relative projection will be $a\{\mathsf{auth}(\mathit{bool}) \,.\, (X \upharpoonright (s, a))\}$. By definition, $X \upharpoonright (s, a) = X$. Hence, this yields the relative type for the continuation of the login branch:

$$G_{\text{login}} \upharpoonright (s, a) = a\{\mathsf{auth}(\mathit{bool}) \,.\, X\} \tag{6}$$

We now turn to the computation of $G_{\text{quit}} \upharpoonright (s, a)$.

$$G_{\text{quit}} \upharpoonright (s, a) = (c \to a : \mathsf{quit} \,.\, \mathit{end}) \upharpoonright (s, a) \tag{7}$$

The exchange between (c, a) does not involve the pair (s, a), so the *ddep* function will be called, with (s, a) as the first argument and $c \to a : \mathsf{quit} \,.\, \mathit{end}$ as the second argument. The *ddep* function will return $\mathit{end} \upharpoonright (s, a)$. By definition, $\mathit{end} \upharpoonright (s, a) = \mathit{end}$. Hence, we get the relative type for the quit branch:

$$G_{\text{quit}} \upharpoonright (s, a) = \mathit{end} \tag{8}$$

Since $G_{\text{quit}} \upharpoonright (s, a) \neq G_{\text{login}} \upharpoonright (s, a)$, the conditions for the first option of the *ddep* function are not fulfilled. However, the server is part of the exchange, thus yielding a non-local choice. The *ddep* function will then return $s!c\{\mathsf{login} \,.\, (G_{\text{login}} \upharpoonright (s, a)), \ \mathsf{quit} \,.\, (G_{\text{quit}} \upharpoonright (s, a))\}$. Considering the dependency returned by the *ddep* function, it is defined and contractive. Hence, the recursive declaration on X is projected as $\mu X \,.\, s!c\{\mathsf{login} \,.\, (G_{\text{login}} \upharpoonright (s, a)), \ \mathsf{quit} \,.\, (G_{\text{quit}} \upharpoonright (s, a))\}$. Substituting the results of Equation (6) and Equation (8), the relative projection of the global type onto the pair (s, a) yields the following:

$$G_{\mathsf{auth}} \upharpoonright (s, a) = \mu X \,.\, s!c\{\mathsf{login} \,.\, a\{\mathsf{auth}(\mathit{bool}) \,.\, X\}, \ \mathsf{quit} \,.\, \mathit{end}\} \tag{9}$$

Van den Heuvel and Pérez define the notion of *relative well-formedness*, which will play a vital role in developing the application for the dynamic verification of protocol conformance. Relative well-formedness is expressed in [23] as the property of global types to have a relative projection defined for every pair of unique participants.

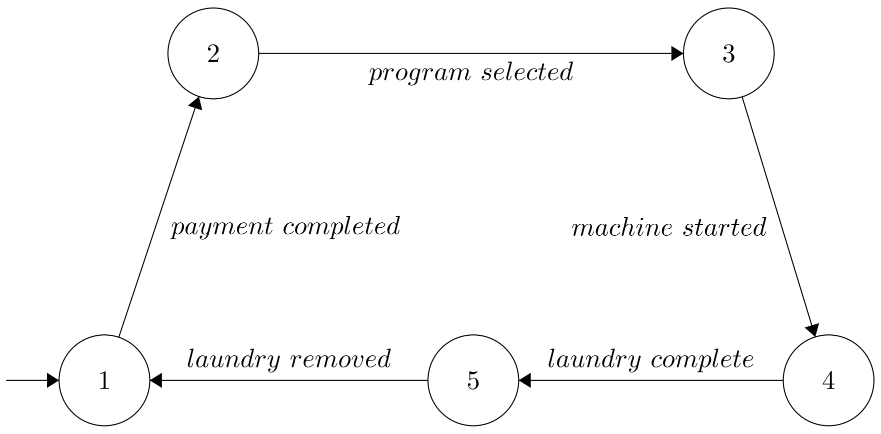


Figure 4: Washing machine as a black-box with five states and transitions between them.

2.4 Finite state machines

Routers employ relative types to check each incoming and outgoing message [23]. Routers will check every observed message internally, and, if it conforms to the given specification, they will forward it to the correct recipient.

Inspecting the structure of relative types, it is clear that the router must be able to verify sequences of messages. Thus, it should be able to store what messages have been verified so far and what messages are permitted in the future. Additionally, relative types support recursive definitions and calls, thus permitting possibly infinite sequences of messages. Hence, we can clearly employ finite state machines to represent relative types.

It is helpful to think of an example to get an intuition of what finite state machines are and how they work. For example, consider laundry centres and the scenario where one individual wants to use a washing machine. The individual must pay for the machine using coins, select the desired program, put the laundry inside the machine, and then start it. After the machine is done washing the given items, the individual must collect them such that someone else can use it. Analysing the perspective of the washing machine in this scenario, we can see that it must first wait for the payment to be completed and then wait for the program to be selected. Afterwards, the machine must wait to be started and then wait for the program to be completed. At last, the machine must wait for the individual to take out the given items. We can reason about the washing machine as a black-box, with the following possible states:

1. payment pending;
2. payment complete, program pending;
3. payment complete, program selected, start pending;
4. machine running;
5. laundry complete.

The washing machine can move to different states once specific events occur. For example, the washing machine can move from the first state to the second state once someone has paid the machine's price. Figure 4 depicts the states of the washing machine and the events that need to happen such that the machine can move on to a different state. States are depicted with circles, and the given events are denoted through arrows. The washing machine is in state 1 initially, and it is depicted with an incoming arrow without an originating state.

Formally speaking, they are represented through the specification of a set of states, an initial state, a set of final states, and transitions to other states whenever certain events occur. Furthermore, finite state machines can transition to previous states, making them an attractive approach to tackling recursive definitions and calls.

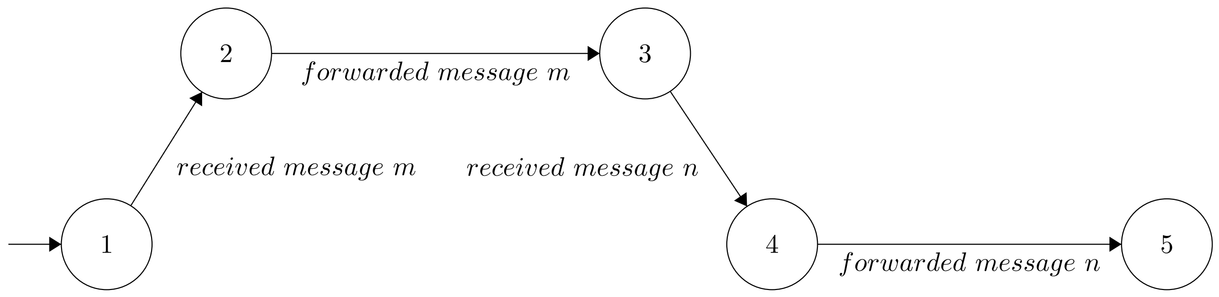


Figure 5: Finite state machine for a router.

To illustrate how finite state machines are helpful for routers, consider the case when a router is supposed to receive a message m from its participant, forward it to the correct recipient, await the receipt of a message n from another router, and then forward it to its participant. We can distinguish five different states for our router:

1. waiting for the receipt of m from the participant;
2. m received;
3. m forwarded to the correct router;
4. n received;
5. n forwarded to the given participant.

Figure 5 illustrates the finite state machine corresponding to the router. A router can inspect its state and observe what events must occur to advance to the next state. If the event that must happen to transition to the next state should be executed by the router (forwarding a message), then the router can take the appropriate action and move on to the next state. Otherwise, if the event depends on a different participant/router (receiving a message), it must wait for the event to happen.

For the purposes of this project, we will use deterministic finite state machines, as each router will only have one transition to a different state.

3 Library implementation

This section details the implementation of the library for monitoring protocol conformance. Section 3.1 first introduces the objectives and the fundamental requirements to put matters into perspective. Thereafter, Section 3.2 describes the technological stack, together with justifications as to why we have chosen each technology and a comparison to suitable alternatives. Section 3.3 defines the architecture of the tool by analysing the library from a vantage point, followed by a description of the protocol specification interface and the router's workings. Finally, Section 3.4 outlines the architecture of participant implementations.

The library is publicly available at:

https://github.com/rares-c/routersMPST/tree/103e01fd899a3ad1b9e884a2d451bb65d42c50e7

3.1 Objectives & Fundamental requirements

This project aims to further the presence of multiparty session types in the realm of dynamic verification methods. To this end, we build a library to verify protocol conformance dynamically by analysing the messages exchanged during execution. Subsequently, the tool provides a means to inspect and establish the conformance to a given protocol in a distributed manner, local to each participant. The tool creates routers for each participant, and they act as middleware for every interaction between the participant and the rest of the network.

To verify protocol conformance dynamically, several core requirements must be satisfied:

1. First and foremost, the library must observe the messages exchanged between the protocol participants. The library then uses these messages to check whether the participants respect the given protocol.

2. Another vital requirement is efficient verification of the observed messages. The library works on top of a network of communicating participants by intervening in message exchanges. Thus, the verification of the observed messages should be as efficient as possible, such that the system's performance does not suffer.

3. Additionally, the decentralised facet of the library implicitly raises additional requirements. Protocol conformance is verified locally for each participant. Thus, deploying the library in any network should not yield radical changes to the implementations of the communicating participants.

4. Finally, the library must verify protocol conformance independently of the source language of the participant implementations.

3.2 Technological Stack

The library we implement employs routers for each participant to verify protocol conformance locally. Therefore, routers need to communicate with other routers and participant implementations. They rely on networking protocols to facilitate this communication. The messages that routers send to other routers and participants need to reach their recipient, and they need to arrive in the order that they were sent. Thus, routers need to make use of a networking protocol that ensures a reliable transmission. For our project, we use the Hypertext Transfer Protocol (HTTP) for the communication between routers and between a router and its corresponding participant. HTTP is built on top of the Transmission Control Protocol (TCP), and it is used in most web services. Moreover, it offers additional security when the Hypertext Transfer Protocol Secure (HTTPS) is used. It is based on the request-response model: one participant makes an HTTP request to an address, and then it receives an HTTP response in return.

Alternatively, we can use TCP directly in our tool to facilitate the transmission of messages. It offers the desired reliability by introducing additional messages for the acknowledgement of the receipt of messages, and a connection set-up phase to ensure that the communicating participants are both ready to transmit messages. TCP is a viable alternative, and it offers a higher degree of customisation. Furthermore, it is slightly more efficient than HTTP, as the transmitted messages do not include any additional data, such as the headers of HTTP messages. Nevertheless, it can be more challenging to implement: after sending a message, the sender must wait for the receiver to acknowledge the receipt of the message. Hence, if the sender is implemented as a single-threaded process, it is effectively blocked until the receiver has acknowledged the message. Consequently, implementing the communication through TCP requires a multi-threaded model to handle sending and receiving messages simultaneously.

The User Datagram Protocol (UDP) offers faster message transfer, but there is no guarantee that the messages arrive at their destination or arrive in order. Compared to TCP, there is no connection set-up phase, and there are no acknowledgements sent after each message. The sender can transmit messages directly to the receiver, regardless of its state. We cannot use UDP in our library, because there is no guarantee that the messages arrive in order or that they arrive at all.

We implement the routers in JavaScript. JavaScript is a dynamically typed language, and it is mainly used in the development of web services. It offers plenty of features that facilitate monitoring particular messages, and its most significant advantage is its ease of use. However, due to the dynamic facet of JavaScript, checking the types of messages poses a challenge. JavaScript is also a weakly typed language, meaning that specific typing rules can be omitted. We can use TypeScript to tackle the challenges mentioned above, as it is strongly typed. Nonetheless, we have chosen to use plain JavaScript for this project, because it provides insight into the challenges of implementing verification methods in dynamic languages. Additionally, JavaScript is widely used in combination with HTTP, so it offers a plethora of ways to implement HTTP communication. Other programming languages are also a possibility, but, as far as we can tell, there are no implementations of session types in JavaScript so far.

JavaScript is mainly used for front-end development. We use Node.js [6] to develop our library, since it provides a runtime environment which can be used to build back-end JavaScript applications. Furthermore, it offers lots of packages that can be used to develop a specific service. Node.js provides the functionality

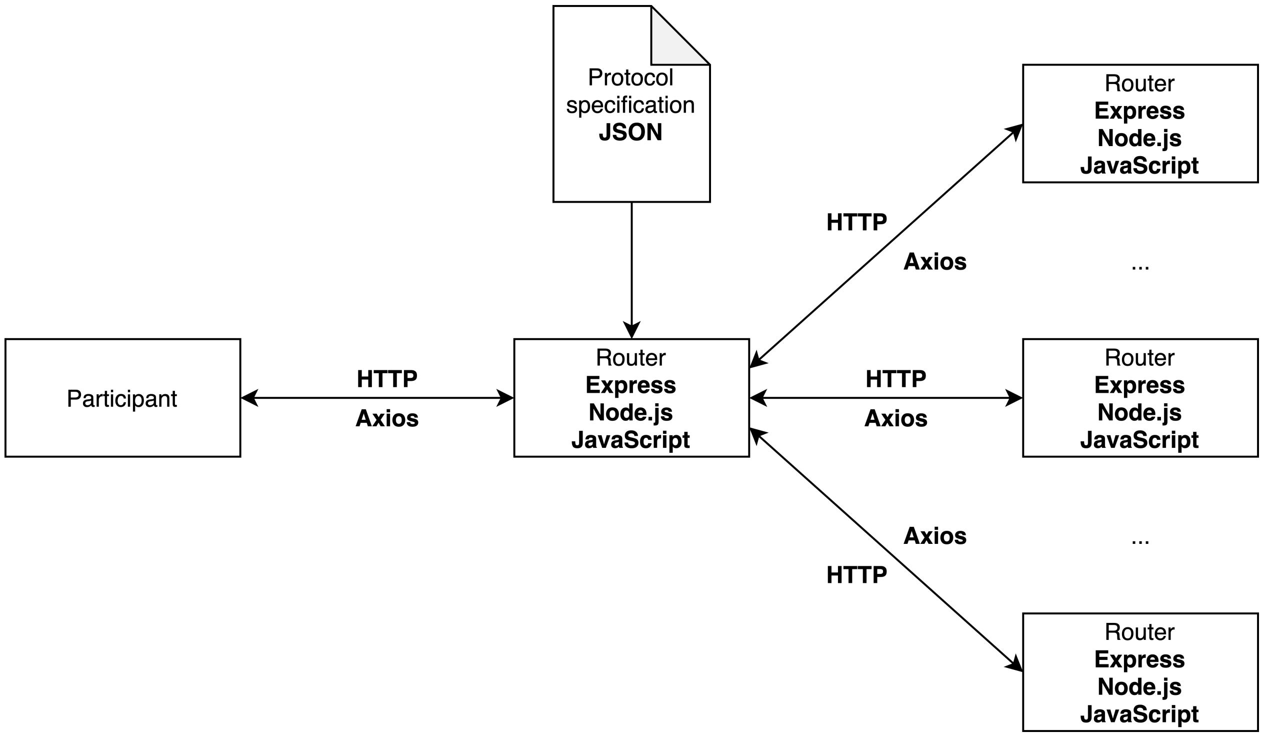


Figure 6: Overview of the technologies used in the library.

needed to create HTTP servers to listen to incoming messages. We use Express [5], a framework for Node.js, to build the HTTP servers on the router side. Express offers a high customisation degree regarding HTTP, and it abstracts a lot of the unnecessary details provided by Node.js for creating HTTP servers. Apart from listening for incoming messages, the routers must forward them to the correct recipients. Therefore, they need to make HTTP requests as well. To this end, we use the package Axios for Node.js. Axios relies on the basic functionality for creating HTTP requests of Node.js; however, it offers a more straightforward interface. To integrate these frameworks and libraries in our application, we use the node package manager (npm). It provides the necessary methods to install several JavaScript frameworks and manage the application's dependencies.

After observing specific messages, routers have to check them against a specification of the protocol. To this end, the protocol is specified using JavaScript Object Notation (JSON), such that the routers parse it efficiently. Furthermore, JSON allows the users to state the required protocol easily, as it follows a clear key-value format. Figure 6 gives an overview of the technological stack, together with the function of each component.

3.3 Library architecture and design

We want to design the library to verify protocol conformance dynamically with modularity and portability in mind. Furthermore, the library aims to be deployable in any network of communicating participants, with minimal changes to the implementations of the participants.

To better understand the separation into different modules, it is helpful to consider the responsibilities of the routers in our library. Before communicating with other routers or participants, a router must first read the specification of the protocol and parse it. Then, a router must obtain a finite state machine corresponding to the actions it must perform, such that its behaviour can be determined. After that, the router must communicate with other routers to ensure that the network is online and ready to transmit messages. Afterwards, the participants can start the protocol, and the routers must intercept every message exchange that occurs in the network, verify it, and then forward it to the correct recipient.

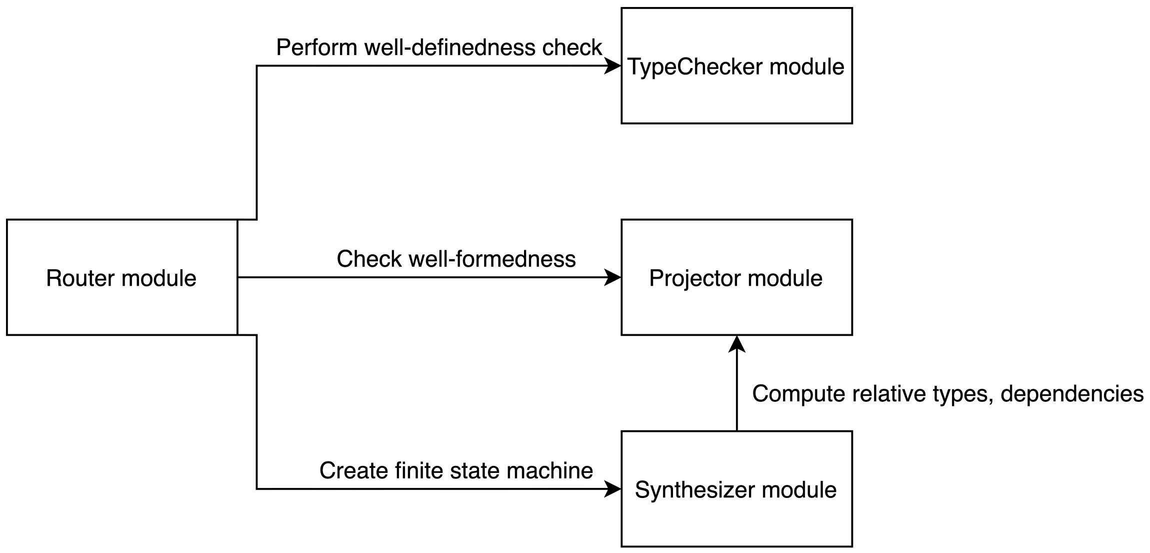

Perform well-definedness check
TypeChecker module
Router module
Check well-formedness
Projector module
Compute relative types, dependencies
Create finite state machine
Synthesizer module

Figure 7: Overview of the modules involved in the library, and their corresponding dependencies.

From the aforementioned requirements of the routers, we can derive five basic needs:

- networking: the routers need to communicate with other routers/participants to exchange messages;
- protocol parsing: the routers need to parse the specification of the protocol;
- relative projections: the routers need to employ relative types in order to determine the messages that should be exchanged with other routers/participants;
- behaviour synthesis: the routers need to define their behaviour from the global type in order to determine what messages must be sent or received;
- message verification: the routers need to check the received message internally to see whether it is the message the router was expecting.

We identify three main components of our library. The `router` module handles the networking and the communication with other routers or participants. It is in charge of receiving messages and forwarding them to the correct participants. Additionally, it deals with verifying the received messages, and it manages the error handling once protocol violations occur. Furthermore, parsing the protocol specification is also the responsibility of the `router` module. The `router` module is the core of our library, as it also handles the calls to the other modules of the library. The `projector` module handles the computation of relative types and relative well-formedness checks. It exposes functions for the computation of relative types, dependencies, and to check the relative well-formedness of a protocol. The `synthesizer` module manages the definition of a router's behaviour by analysing the protocol specification from the perspective of the corresponding participant. It uses the router synthesis algorithm defined in [23], and it depends on the `projector` module for the computation of relative types and dependencies. The `synthesizer` module exposes only a single function that handles the creation of a finite state machine corresponding to the behaviour of a specific router. Additionally, the library employs another module to perform a well-definedness check on the specification of the protocol. The `typechecker` module checks the global type to ensure that all participants used in exchanges are defined beforehand, and it checks whether recursive calls do not use undefined recursion variables and recursive definitions do not use previously defined variables. Figure 7 shows an overview of how the modules interact. Modules are represented by rectangles, and an arrow from a module A to another module B labelled x denotes the use of functionality x by A provided by module B.

3.3.1 Protocol specification interface

The specification of the protocol is represented as a global type. Thus, it needs to support message exchanges, recursive definitions, recursive calls, the end of the protocol, and sequencing. We analyse each requirement individually to determine what attributes are needed and how they are stored in JSON. The protocol specification interface makes use of `global_interaction` objects. Each `global_interaction` object has a field `type` to distinguish between message exchanges, recursive definitions, recursive calls and the end of the protocol.

- In case of an exchange, the interface should capture the sender, the receiver, and all possible labels, types, and continuations. The sender and the receiver are specified using strings, whereas the labels, types, and corresponding protocol continuations are specified using dictionaries. The set of dictionary keys coincides with the set of labels. Each label is associated with an object specifying the type of the value accompanying the label and the continuation of the protocol. The types of each value are specified as strings, and the continuation of the protocol is specified as a reference to another `global_interaction` object.
- In case of a recursive call, the interface should capture the recursion variable, specified as a string.
- In case of a recursive definition, the recursion variable and the continuation of the protocol need to be tracked. Therefore, the recursion variable is specified as a string, whereas the continuation of the protocol is specified as a reference to another `global_interaction` object.
- For the end of the protocol, the interface does not need additional attributes.
- For sequencing multiple interactions, the protocol specification interface uses linked `global_interaction` objects: each `global_interaction` object has at least one reference to another `global_interaction` object, with the exception of recursive calls or the end of the protocol. Consequently, the interactions are nested and only accessible through their parent. This ensures a strict ordering and prevents routers from accessing random interactions.

The protocol specification interface contains additional details besides the global type. As the routers use HTTP to communicate with other routers and participant implementations, they need the address of every other router and the address of their corresponding participant. To this end, a dictionary associating each participant with an address is added to the protocol specification. Additionally, the port on which the router listens to incoming messages is included in the protocol specification. Finally, routers require the name of the participant they encapsulate in order to define their behaviour. Hence, the protocol specification interface also includes the name of the participant corresponding to the current router, making each JSON file specific to a certain participant.

3.3.2 Router initialisation and execution

The `router` module represents the core of the library. It is responsible for parsing the protocol, the communication with other routers or participants, and handling protocol violations. Furthermore, it offers the functionality needed to deploy the routers and execute the actions encapsulated by their finite state machines once all participants are ready to start the protocol.

Before commencing the communication, the tool must execute the router initialisation process. Figure 8 depicts the initialisation procedure of the routers, and it offers hyperlinks encoded as numbers to the paragraphs of this section that describe each particular step. Routers require access to the specification of the protocol, such that they can encode their behaviour as finite state machines. Thus, the module first reads the protocol and then parses it. Since the user specified the protocol in JSON, parsing it is straightforward for a JavaScript module.

Well-definedness check As the data provided by the users is prone to errors, the initialisation process employs a well-definedness check procedure to verify whether the protocol has any errors. The verification is realised through the `typechecker` module to ensure modularity.

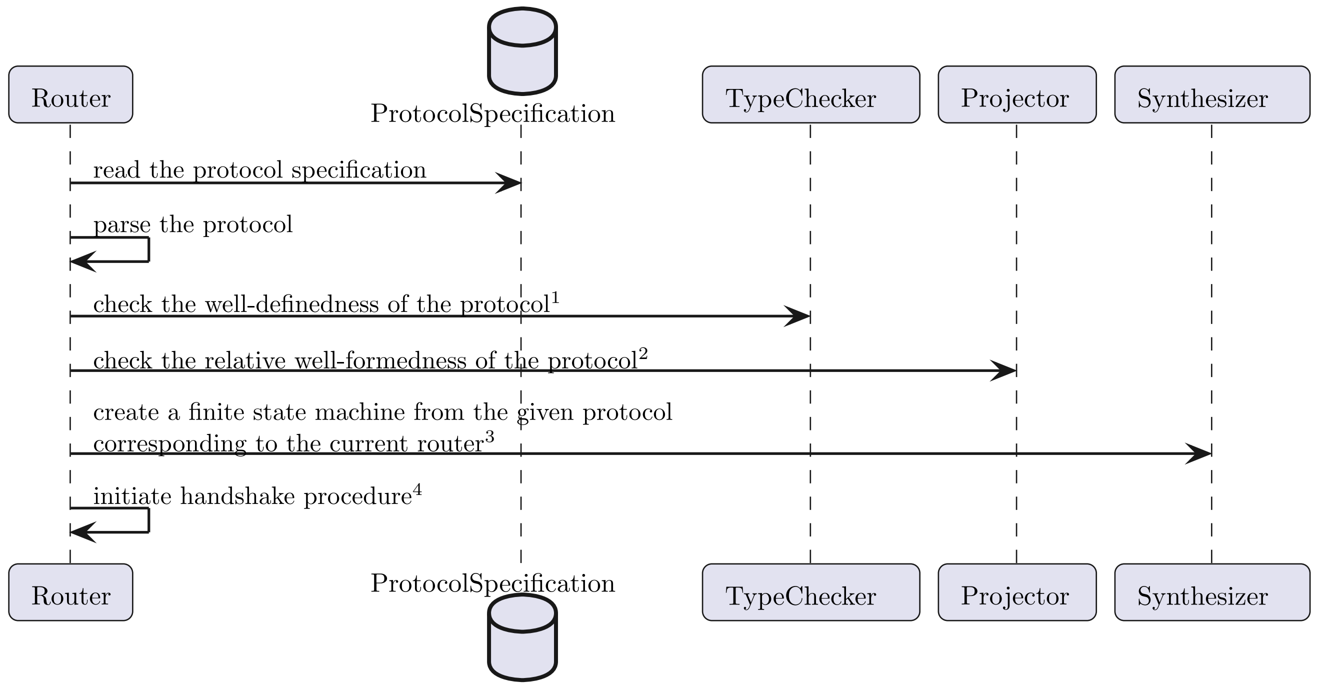

Router
ProtocolSpecification
TypeChecker
Projector
Synthesizer
read the protocol specification
parse the protocol
check the well-definedness of the protocol1
check the relative well-formedness of the protocol2
create a finite state machine from the given protocol
corresponding to the current router3
initiate handshake procedure4
Router
ProtocolSpecification
TypeChecker
Projector
Synthesizer

Figure 8: Router initialisation process illustrated as a sequence diagram.

The `typechecker` module adds a layer of security to the implemented library by traversing the global type in a depth-first manner and checking for well-definedness errors. Consequently, the initialisation process of the routers is more robust, as the tool informs the users of potential well-definedness errors.

Algorithm 1 displays the algorithm used to detect well-definedness errors. It traverses the global type recursively, and it keeps track of the recursion variables defined so far and the list of participants. It checks for three well-definedness errors:

- Exchanges involving participants not defined beforehand, thus not included in the participant list;
- Recursive calls using an undefined recursion variable;
- Recursive definitions using a recursion variable that is already in use.

Relative well-formedness verification The global type is also checked for relative well-formedness as defined in [23], to ensure that every pair of distinct participants has a relative projection defined. The relative well-formedness check happens through the `projector` module.

In order to find a suitable internal representation for the relative types, one must inspect the syntax of relative types given in Section 2.3. We observe that the internal representation for relative types should support six different cases: exchanges, input and output dependencies, recursive definitions, recursive calls, and the end of the protocol.

We define `interaction` objects meant to represent the projection of an interaction onto a pair of different participants. Each `interaction` object has a field `type` to distinguish between the aforementioned cases. Furthermore, depending on the type of the object, additional attributes are used to encapsulate the essential information of the relative type. We represent sequencing through the use of an attribute which keeps track of the continuation. Hence, the relative type corresponding to a given pair consists of a chain of `interaction` objects. Figure 9 gives an overview of the types of the `interaction` objects and their corresponding attributes.

Algorithm 1 Well-definedness verification of a global type.

```
Require: Global type $G$, list of recursion variables defined so far $vars$, list of participants $parts$
Ensure: Global type does not contain any undefined participants, recursive calls using undefined variables,
    recursive definitions using previously defined variables, or error
1:  if $G_{\mathsf{type}}$ = EXCHANGE then
2:      if $G_{\mathsf{sender}} \notin parts \lor G_{\mathsf{receiver}} \notin parts$ then
3:          Well-definedness error: undefined receiver or sender
4:      end if
5:      for label $\in G_{\mathsf{branches}}$ do
6:          Apply the well-definedness algorithm to the continuation of the label, $vars$ and $parts$
7:      end for
8:  else if $G_{\mathsf{type}}$ = RECURSIVE CALL then
9:      if $G_{\mathsf{recursionVariable}} \notin vars$ then
10:         Well-definedness error: recursive call to undefined variable
11:     end if
12: else if $G_{\mathsf{type}}$ = RECURSIVE DEFINITION then
13:     if $G_{\mathsf{recursionVariable}} \in vars$ then
14:         Well-definedness error: recursive definition using previously defined variable
15:     else
16:         Apply the well-definedness algorithm to the protocol continuation, $vars + G_{\mathsf{recursionVariable}}$ and
    $parts$
17:     end if
18: end if
```

Behaviour definition After the routers verify the global type for well-definedness and relative well-formedness, they obtain a finite state machine corresponding to the actions they must perform. They delegate the creation of the finite state machine to the `synthesizer` module, which employs the router synthesis algorithm defined in [23].

The finite state machine identifying a specific router should express the actions it must perform (send or receive messages) and their corresponding ordering. Additionally, the states of the finite state machine should facilitate an efficient verification of the observed messages.

The algorithm to create a finite state machine defining the behaviour of a router happens in two passes. In the first pass, the algorithm generates the router process by expressing what messages the router is supposed to receive and send. It closely follows the router synthesis algorithm defined in [23]. In the second pass, the algorithm identifies the states that correspond to the continuations of the recursive definitions, and it replaces recursive calls with references to those states. Consequently, recursive calls and definitions are

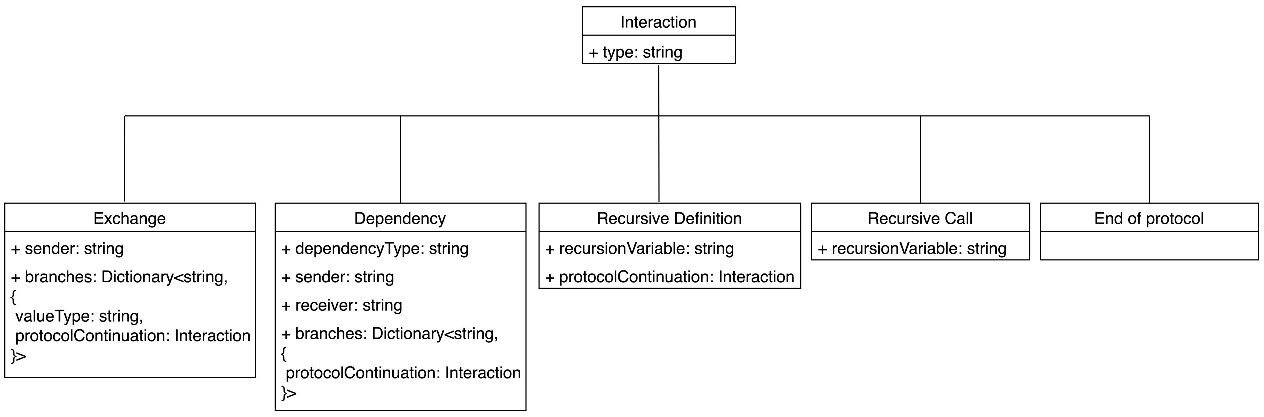

Interaction
+ type: string
Exchange
+ sender: string
+ branches: Dictionary<string, { valueType: string, protocolContinuation: Interaction }>
Dependency
+ dependencyType: string
+ sender: string
+ receiver: string
+ branches: Dictionary<string, { protocolContinuation: Interaction }>
Recursive Definition
+ recursionVariable: string
+ protocolContinuation: Interaction
Recursive Call
+ recursionVariable: string
End of protocol

Figure 9: Overview of the `interaction` objects types and their attributes.

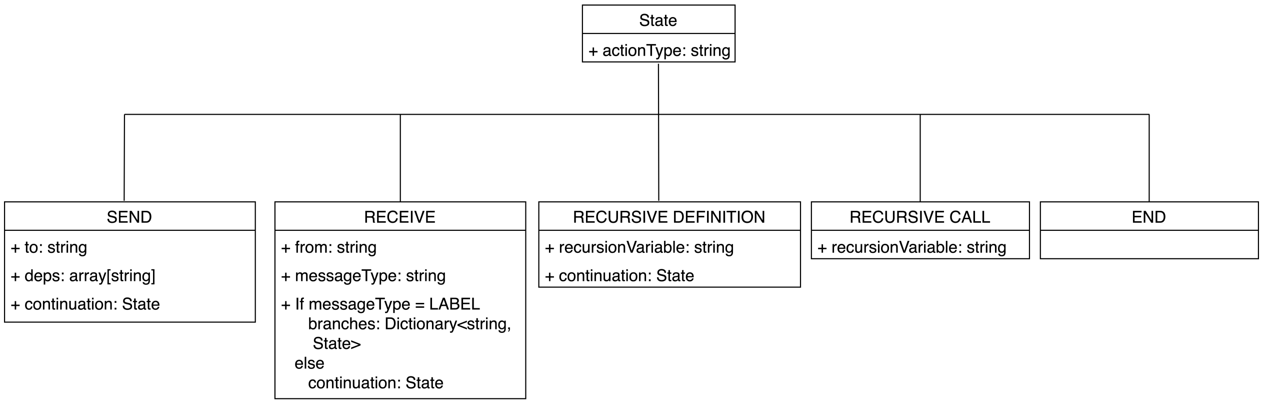


Figure 10: Types of states involved in the creation of the finite state machine defining a router's behaviour.

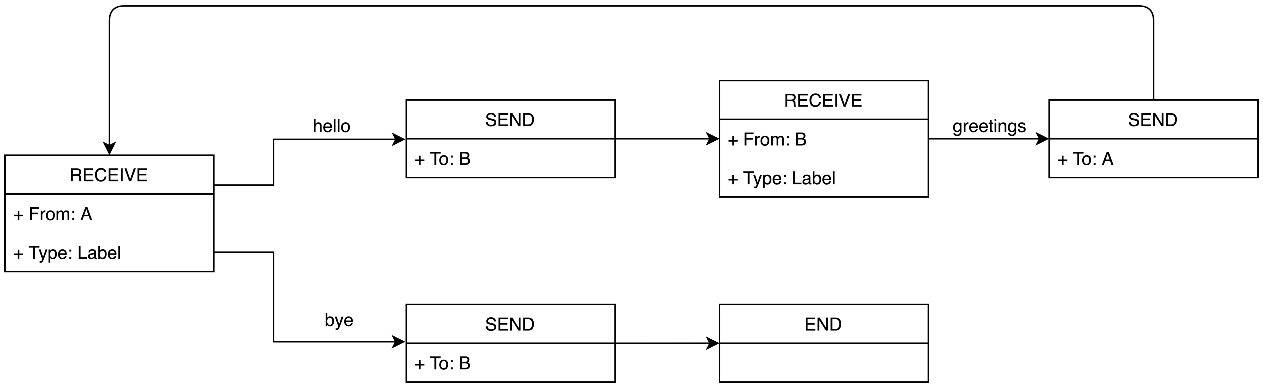


Figure 11: Finite state machine identifying the router of A from Equation (10).

removed altogether from the final representation of the finite state machine. Figure 10 shows the different types of states involved in creating the finite state machine.

The first pass traverses the global type recursively, generating appropriate states at each step. To understand the intuition behind the algorithm, it is helpful to consider an example. Suppose that we are given the following global type, and we are interested in computing the finite state machine for the router of participant A.

$$\mu X \,.\, A \to B : \{\mathsf{hello} \,.\, B \to A : \mathsf{greetings} \,.\, X, \quad \mathsf{bye} \,.\, end\} \tag{10}$$

The global type from the above represents a simple protocol for two participants greeting each other. It starts with a recursive definition on X, followed by an exchange from A to B labelled hello or bye. If A chooses the hello label, B replies with a label greetings, and the protocol loops to the beginning. On the other hand, if A chooses the bye label, the protocol terminates. Hence, inspecting the protocol from A's perspective, it can send a label hello to B and receive a label greetings in return zero or more times, or it can send a label bye to B, and the protocol finalises. Thus, the router of A, which is supposed to intervene in every message exchange, should perform the following actions zero or more times: receive a label hello from A, forward it to B, receive a label greetings from B, and forward it to A. Then, it should receive a label bye from A, and forward it to B. We can represent the sequence of actions of router A through the finite state machine given in Figure 11.

The first pass of the algorithm proceeds in a case-by-case analysis on the structure of the first interaction of the global type:

- If the first interaction is an exchange between two participants, the algorithm computes a set of participants that depend on the given exchange. Then, it checks whether the router's participant is involved in the exchange:

  – If it is involved in the given exchange, the algorithm returns a receive state, including all possible labels of the exchange and the corresponding sender. Additionally, each label offers a transition to a send state, where the router forwards the given label to the receiver of the exchange and every dependency. Next, the algorithm checks the type of the value accompanying each label. Unit types are not transmitted, as they would introduce superfluous messages in the system. On the other hand, string, integer, real or boolean values entail two additional states: one receive state for the receipt of the value and one send state for transmitting the value to the appropriate receiver. Then, the algorithm carries on with the continuation of the protocol for each particular branch.
  – If the router's participant is not involved in the exchange, the algorithm checks whether it is dependent on the sender or the receiver. If there is a dependency on the sender or the receiver, the algorithm yields a receive state for the labels of the exchange. Consequently, each label leads to a send state, where the router forwards the dependency label to its participant. If there is a dependency on both the sender and the receiver, the algorithm generates an additional receive state, such that the router receives the same label twice. Thereafter, the algorithm is applied to the continuation of each particular branch. If the router's participant does not depend on the sender or the receiver, the algorithm is applied to the continuation of the first branch, as the current exchange does not influence the router's participant.

- If the first interaction is a recursive definition, the algorithm checks whether the router's participant is involved in the continuation of the protocol. To this end, the algorithm computes the relative projection of the protocol's continuation with every other participant. If there is at least one relative projection that does not yield the end of the protocol, the algorithm generates a state identifying a recursive definition, and it proceeds to compute the finite state machine for the continuation of the protocol. Otherwise, it returns an end state.
- If the first interaction is a recursive call, the algorithm returns a state identifying the recursive call.
- At last, if the first interaction is the end of the protocol, the algorithm returns an end state.

The second pass of the algorithm traverses the finite state machine recursively, and it removes recursive definitions and recursive calls. Upon encountering a recursive definition, the algorithm removes the current state, linking the parent state with the continuation of the protocol. It stores the current recursion variable, such that it can be associated with a reference to the next send or receive state. Upon encountering a recursive call, the current state is replaced by a reference to the send or receive state associated with the given recursion variable. Note that each recursive definition is guaranteed to use a variable that has not been defined before, and recursive calls use recursion variables defined previously, due to the well-definedness check employed beforehand.

The router handshake procedure Upon defining the behaviour of a specific router as a finite state machine, the initialisation process advances to the handshake step, where the routers wait for the entire network to be set up and ready for transmission.

The handshake operation is meant to coordinate the routers and participants to start the protocol. Consider a scenario involving three participant implementations and three routers. Two participant implementations are online and ready to commence the communication, and all three routers are online and ready as well. However, one participant is not online yet, so the protocol should not start. If the tool would not use the handshake procedure, the participants may start the communication. Thus, the participant that has not been started yet is not able to participate in the given interactions, resulting in a potential loss of messages and possible protocol violations.

The handshake procedure aims to provide the user with the opportunity to start the components of the network in no particular order. The library aims to be deployable on top of an existing network of participant implementations. To this end, we impose the constraint that all participant implementations must be started first, and then the routers. As such, the participant implementations wait for the routers to signal that the network is ready.

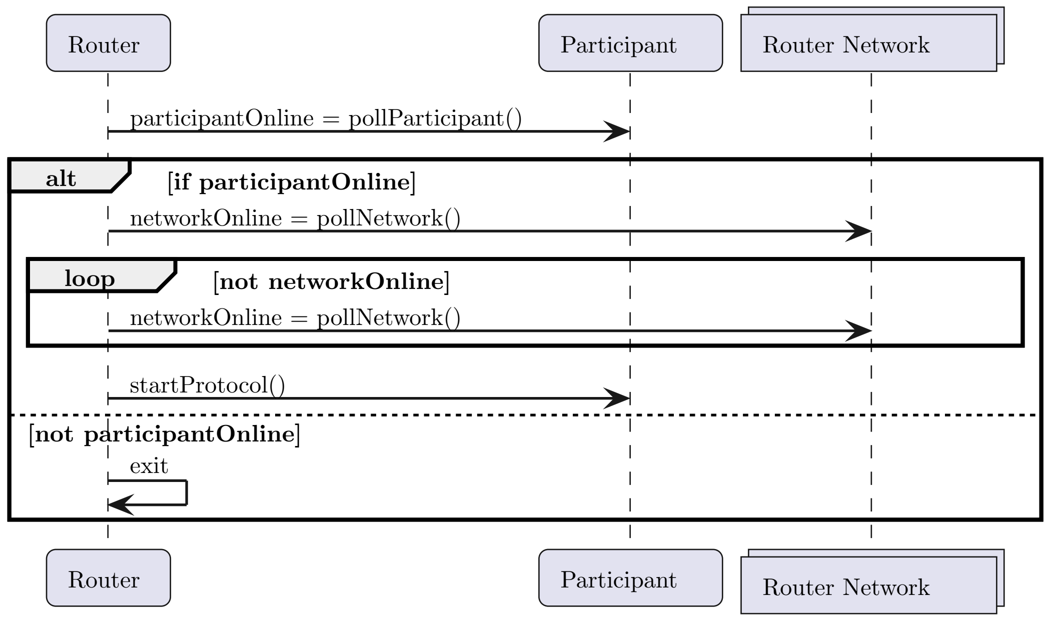


Figure 12: Overview of the router handshake procedure.

As Node.js is single-threaded and handles incoming messages asynchronously, we must take special care to avoid running into deadlocks (routers waiting indefinitely for other routers). Thus, we place certain constraints on how the routers coordinate with one another and with the participant implementations. The handshake procedure consists of three steps enumerated below. Figure 12 illustrates the handshake protocol as a sequence diagram.

1. Routers first attempt to communicate with their corresponding participant implementation to assess whether it is online or not. If their corresponding participant implementation is not online or not available, the routers warn the user about it and exit.

2. If their corresponding participant implementations are online, the routers attempt to communicate with every other router to assess the state of the network. If another router is reachable, then there is a guarantee that it is online and its participant implementation is online as well. A router polls every other router until they are all available.

3. At last, all routers are available, and all participant implementations are online and waiting for the signal from their corresponding routers. Thus, the routers let the participant implementations commence the protocol.

Communication in the network   If the handshake procedure succeeds, the participants can start the communication. They send messages to their corresponding routers through HTTP, and the routers verify and forward them to the correct recipient.

Each HTTP request targets a specific address and route. For instance, two requests can target the address `http://localhost`, and the routes `/route1/` and `/route2/`, yielding `http://localhost/route1/` and `http://localhost/route2/`. Requests use different methods to denote the actions that must happen on the recipient's side. We distinguish between two types of requests used in our program: POST requests that alter the state of a router or participant, and GET requests that do not modify the state of a router or participant. Routers and participants treat requests meant to transmit messages of the protocol as state altering, and requests that check the status of a router or participant as non-altering. For transmitting messages of the protocol, the routers and participants use POST requests on the root path `/`. Routers and participant implementations also support POST requests on the path `/api/violation/` such that other routers can inform them whenever a protocol violation occurs. GET requests on the route `/api/alive/`

are supported by both routers and participants to test whether their corresponding recipients are online. Additionally, participant implementations should support POST requests on the route `/api/alive/` such that routers can signal them to begin the protocol.

Message structure Once the routers receive a message, they check it internally to see whether it conforms to the given specification, and then they forward the message to the actual recipient. The internal representation of the messages must allow the routers to verify them efficiently. Inspecting the syntax of multiparty session types, it becomes apparent that routers need to check whether the observed messages are intended for the correct recipients, whether they originated at the correct participant, and whether the type of the message conforms to the given specification. The exchanged messages must specify the sender, the receiver, and the payload of the given message in order to satisfy the aforementioned requirements. The routers encode the messages in JSON such that other routers or participants can easily access them. Following is an example of a message satisfying the previous requirements: participant a created a message labelled *password* for b.

```
{
  "sender": "a",
  "receiver": "b",
  "payload": "password"
}
```

The aforementioned scheme for the representation of the messages exchanged in the network limits the value types available for the participants to strings, integers, real numbers and booleans. Moreover, it introduces a security risk, as routers do not verify the authenticity of the messages and their corresponding senders. This problem could be solved by introducing a router authentication mechanism to guarantee the identities of other routers and participants.

Finite state machine execution The finite state machine defines the behaviour of the router, in terms of the actions that a router must complete. It specifies the type of action a router must complete (receive or send) and the specifics of its corresponding message.

Whenever a router receives a message, it executes the actions and checks defined in the current state according to Algorithm 2. It is important to note that whenever a message arrives at a router, it is always in a receive state. Additionally, receive states alternate with send states, as a router must forward every label/value it receives to the correct recipient. If a participant depends on both the input and the output of an exchange, its router receives the dependency label from both the sender and the receiver of the exchange, but it forwards the label to its participant only once.

Protocol violation handling Protocol violations occur whenever the received message does not match the expected message in a router. Whenever such a violation happens, the router throws an error, which is caught and delegated to an appropriate handler. Handlers are defined in the router, and they determine how routers react to protocol violations. Routers support two types of handlers by default: panic mode, and recovery mode. Panic mode is the most basic type of error handling: it informs the user about the error, it notifies every other router and participant implementation about the violation and exits the process. This type of handler is particularly useful in protocols where no errors are admissible. For example, one can use panic mode in banking systems, where any errors should terminate the communication immediately to prevent erroneous transactions or exchanges. Recovery mode is more complex than panic mode. Whenever a protocol violation occurs, the router tries to revert to the last receive state, such that the sender of the message gets the opportunity to correct their mistake. Recovery mode tolerates a certain degree of errors. However, it relies on the assumption that the sender of the incorrect message retries to send the correct one. If the participant is unable to acknowledge that the transmitted message was violating the protocol, recovery mode is rendered useless, as the correct message is not retransmitted. It is important to note that the routers always try to revert to the last receive state, such that the participant implementations have the chance to resend the incorrect message. If the routers were to revert to the last send state, they would forward the message encoded in the state, wrongfully introducing redundant messages in the system.

Algorithm 2 Execution of the finite state machine identifying the behaviour of a router.

```
Require: Message $M$; Current FSM State $S$
Ensure: Message verified and forwarded, otherwise protocol violation
 1: if $M_{\mathsf{sender}} \neq S_{\mathsf{from}}$ then
 2:     report protocol violation: actual sender does not match the expected sender
 3: else if $S_{\mathsf{messageType}} = \mathsf{label}$ and $\mathsf{type}(M_{\mathsf{payload}}) \neq \mathsf{string}$ then
 4:     report protocol violation: expected label, received value
 5: else if $S_{\mathsf{messageType}} = \mathsf{label}$ and $M_{\mathsf{payload}}$ not in $S_{\mathsf{possibleBranches}}$ then
 6:     report protocol violation: unknown label
 7: else if $S_{\mathsf{messageType}} \neq \mathsf{type}(M_{\mathsf{payload}})$ then
 8:     report protocol violation: expected type of the message does not match the actual type
 9: end if
10: $S \leftarrow \mathsf{nextState}(S)$        ▷ Received the expected message, conditions satisfied, transition to next state
11: if $S_{\mathsf{action}} = \mathsf{end}$ then
12:     exit router                                  ▷ End of the protocol
13: else if $S_{\mathsf{action}} = \mathsf{receive}$ then
14:     return                    ▷ Next state is a RECEIVE state, no need to forward the message
15: end if                ▷ Next state is a SEND state, forward the message to the correct recipients
16: if $M_{\mathsf{receiver}} \neq S_{\mathsf{to}}$ then
17:     report protocol violation: actual receiver does not match the expected receiver
18: end if
19: Forward message $M$ to its actual receiver
20: for $\mathsf{dependency} \in S_{\mathsf{dependencies}}$ do
21:     Forward message $M$ to $\mathsf{dependency}$
22: end for
23: $S \leftarrow \mathsf{nextState}(S)$
24: if $S_{\mathsf{action}} = \mathsf{end}$ then
25:     exit router                                  ▷ End of the protocol
26: end if
```

3.4 Participant implementation architecture

This section describes the architecture of participant implementations in JavaScript and Node.js. As previously said, the participant implementations can be created using any language or framework. We have chosen to discuss the potential implementation of a participant in Node.js. Moreover, the library provides a template written in JavaScript that can be used to develop participant implementations. In the future, templates in different programming languages can be added.

By default, participant implementations need to support POST requests on the root path `/` for receiving messages from their routers. Furthermore, they need to support two more routes in order to aid the router handshake process: GET requests on the path `/api/alive/` to signal the router that the participant implementation is online, and POST requests on the path `/api/alive/` to let the participant implementation know that the communication can start. Participant implementations also need to support POST requests on the path `/api/violation/`, such that routers can inform them about any violations. Since we are using HTTP as a transport mechanism, every request entails an empty response.

The only vital requirement of the participant implementations is the HTTP server providing the four different routes mentioned previously. The participant implementations are free to handle different sequences of messages however they desire. Nevertheless, the solution of using finite state machines to encode the messages that participant implementations expect to receive and the actions they must perform offers a way to tackle asynchronous HTTP request handling.

As Node.js handles incoming requests asynchronously, the participant implementations can use finite state machines to encode the sequences of messages they must receive and the actions they must perform. There is a state in the finite state machine for each message the participant implementation expects to receive. The finite state machine transitions to new states depending on the messages it receives. If the participant implementation can receive multiple labels in the current message, it can transition to different states based

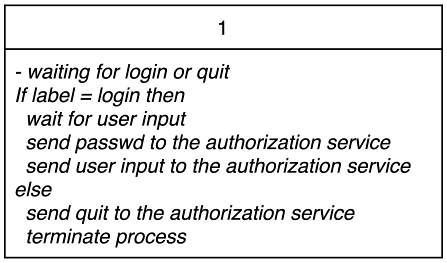

1
- waiting for login or quit
If label = login then
wait for user input
send passwd to the authorization service
send user input to the authorization service
else
send quit to the authorization service
terminate process

Figure 13: Finite state machine corresponding to the messages the client must receive and its actions.

on the label it received. Otherwise, there may only be one transition to a new state.

4 Client-Server-Authorization protocol

This section describes the implementation of the Client-Server-Authorization protocol by Van den Heuvel and Pérez [23], adapted from [22]. It dives into the three participant implementations, how they interact, and the results that were obtained by deploying the implemented library in two different scenarios: when all the participant implementations respect the given protocol, and the scenario when one of the participant implementations produces a protocol violation by transmitting an incorrect label. The global type for the Client-Server-Authorization protocol is the following:

$$G_{\mathsf{auth}} = \mu X \,.\, s \to c : \{\mathsf{login} \,.\, c \to a : \mathsf{passwd}(str) \,.\, a \to s : \mathsf{auth}(bool) \,.\, X, \quad \mathsf{quit} \,.\, c \to a : \mathsf{quit} \,.\, end\} \tag{11}$$

As Node.js handles incoming requests asynchronously, special care must be taken to encode the sequences of actions that must be performed. The actions a participant implementation must perform are represented as a finite state machine to support sequences of different messages and recursive calls or definitions. The states of the finite state machine contain specific actions, such as sending a label and/or a value, and the participant implementation transitions to new states once all the actions are performed. Whenever the participant implementations receive a message, they check the state they are in and perform the appropriate actions. Moreover, if they need to transmit one or multiple messages, they initiate the requests to other participants, and then move on to a new state. Consequently, the participants no longer need to block until receiving a message, and they can naturally handle recursion through transitions to previous states.

In the implemented protocol, the server asks the client to log in until the authorization is successful. The client waits for the user to input the password, and it sends to the authorization service a label passwd, and the given password as a string. The authorization service then checks whether the received password matches its secret password, and it sends to the server a label auth and a boolean value to reflect it. The server checks whether the authorization was successful or not. If the client failed to log in, the server asks the client to perform the whole procedure again by sending another login label to the client. If the authorization succeeded, the server asks the client to quit. Thereafter, the client sends to the authorization service a quit label, and the protocol finalises.

Figure 13 displays the finite state machine identifying the client implementation, whereas Figure 14 shows the finite state machine corresponding to the server implementation. Figure 15 displays the finite state machine identifying the authorization service implementation. Upon receiving a message, each participant implementation checks the state it is in and takes the appropriate actions. The initial action of the server implementation (sending a login label to the client) is omitted, as the server should only send it after receiving the confirmation from its router that the network is ready.

The participant implementation of the server contains the functionality needed to transmit the initial choice in the method that handles POST requests on the path `/api/alive/`, whereas the finite state machines are encapsulated in the method responsible for POST requests on the root path `/`. The separation of the participants' initial actions and their finite state machines is crucial, as the participants should only start the transmission after receiving the signal that the network is ready from their routers.

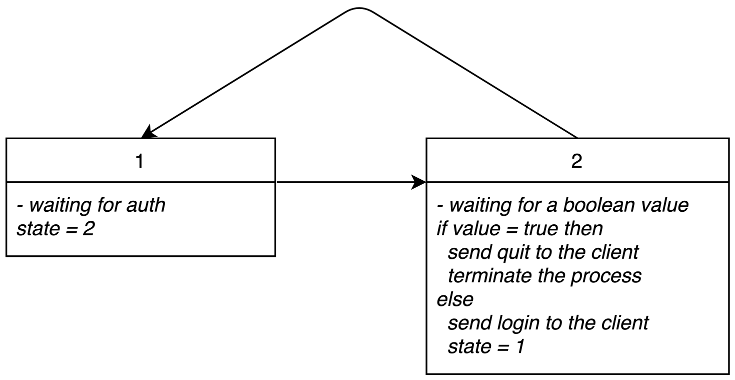


Figure 14: Finite state machine corresponding to the messages the server must receive and its actions.

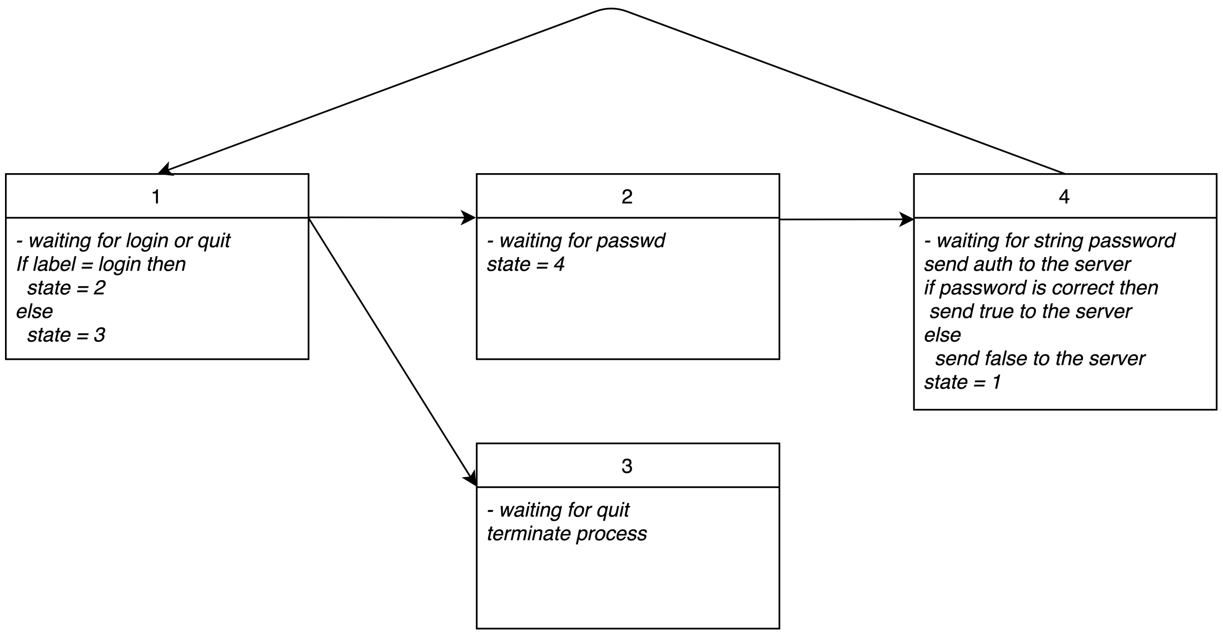


Figure 15: Finite state machine corresponding to the messages the authorization service must receive and its actions.

We can now execute the participant implementations and deploy the implemented library. We observe the results that are obtained in two different scenarios: when all the parties adhere to the given protocol, and when the client sends a `password` label instead of the `passwd` label.

Figure 16 displays the scenario when all the participant implementations adhere to the Client-Server-Authorization protocol, whereas Figure 17 illustrates the scenario when a protocol violation is produced by the client. In both figures, the first column of terminal windows displays the output of the implementations, whereas the second column displays the output of the router processes. The client implementation and router is displayed on the first row, whereas the server and the authorization service are displayed on the second, respectively third rows. In both cases, the routers first perform the handshake procedure before commencing the transmission.

In the first scenario, the client first sends an incorrect password to the authorization service and it is asked to log in again. Thereafter, it sends the correct password, and the protocol finalises. In the second scenario, the client sends the password label to the authorization service's router instead of passwd, resulting in a protocol violation at the client's router.

5 Networks of incompatible endpoints

This section illustrates the compatibility of the tool with networks involving third-party participant implementations that are not directly compatible with routers. It first describes an example protocol that is used to demonstrate the applicability of the tool, and then it details a potential solution for dealing with incompatible participants. Thereafter, the results of applying the tool in two scenarios are presented: when all participants adhere to the given protocol, and when the incompatible participant produces a protocol violation.

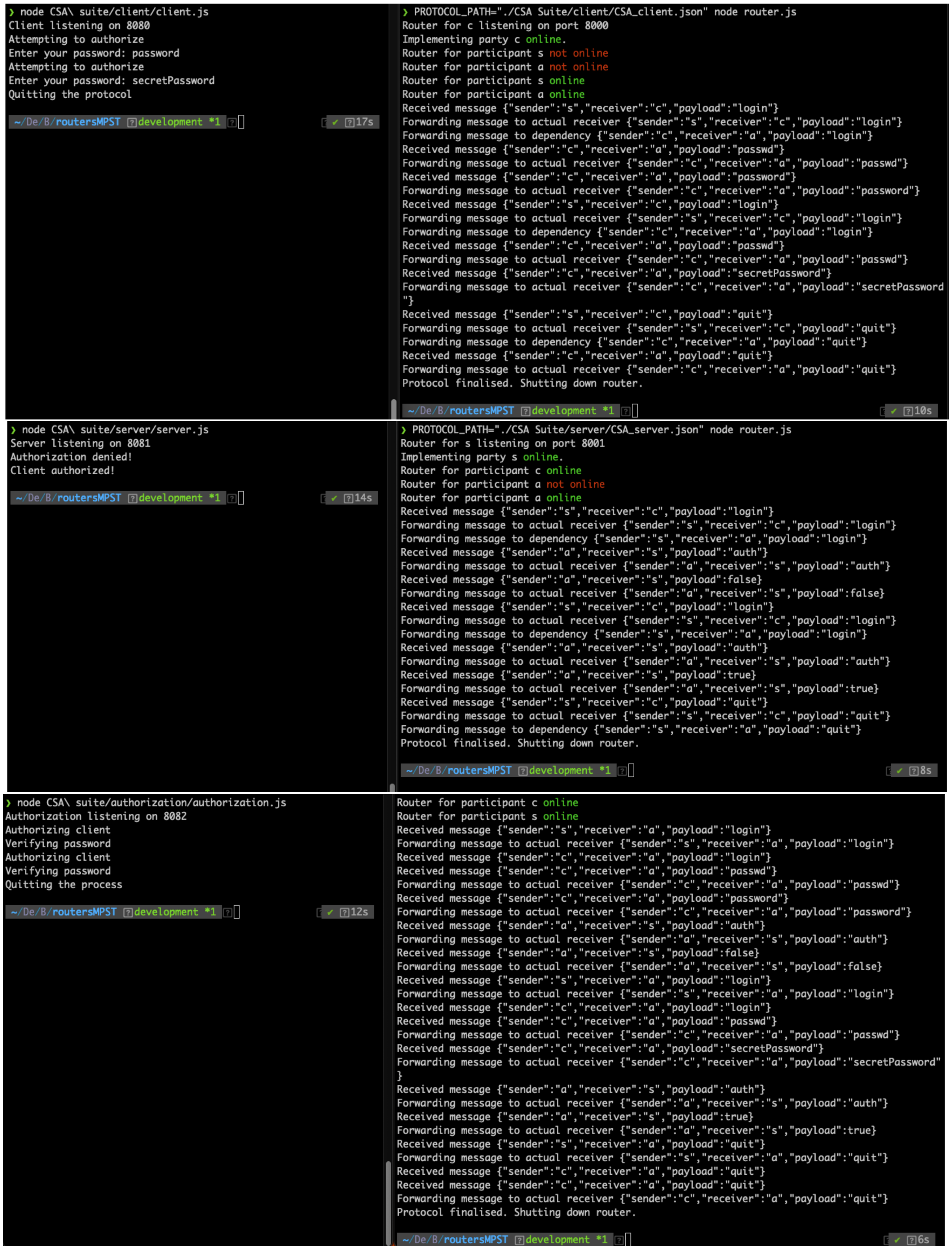
The protocol that we use to test our tool involves three participants: a client (c), a database server (d), and a weather service (w). The goal of the protocol is to find the temperature in a city using its coordinates and the weather API provided by [21]. The client first provides the API key to the weather service. Thereafter, the client asks the database server for the coordinates of a given city. If the selected city is stored in the database, the server returns its coordinates. Otherwise, it tells the client that the requested city is not available. Thereafter, if the client received a set of coordinates, the client proceeds to request the weather at those specific coordinates. Otherwise, it tries to query the database for another city or end the protocol. The weather service responds with the temperature, and the client is offered the choice to request weather information for another city or to quit. The global type of the protocol is as follows.

$$\begin{aligned} G_{\mathsf{weather}} = c \to w : \mathsf{key}(str) \,.\, \mu X \,.\, c \to d : \{&\mathsf{city}(str) \,.\, d \to c : \{\mathsf{coordinates}(str) \,.\, c \to w : \mathsf{coordinates}(str) \\ &.\, w \to c : \mathsf{temperature}(real) \,.\, X, \;\; \mathsf{missingCity} \,.\, X\}, \;\; \mathsf{quit} \,.\, end\} \end{aligned} \tag{12}$$

We consider an implementation of this protocol where the client and database are compatible with routers, but the weather service is not, as it is a third-party API whose behavior we cannot change. We implement the client such that it prompts the user for an API key, then it asks the user one or more times to input a city or to quit. If the client inputted a city, the client queries the database server for the coordinates of that specific city and then it asks the weather API for the weather at those exact coordinates. We implement the database such that it stores the coordinates of 6 cities: Groningen, Amsterdam, Zwolle, London, Bucharest, and Bern. If the client asks the database for the coordinates of a city, it first checks whether the given city is stored inside the database. If it is, the database responds with the coordinates of the city. Otherwise, it tells the client that the city is missing from the database.

In order to solve the compatibility issue of the weather API with the routers, we create a new component meant to wrap the API. The wrapper therefore handles the communication with the weather API, and it translates all requests in the format needed by the API. In doing so, we implement the role of w in G_{weather} as an interface to the weather API. We can abstract the wrapper and the weather API as a whole, yielding a participant implementation compatible with the routers. Figure 18 illustrates the network of communicating participants before and after the routers and the wrapper were deployed.

The weather wrapper communicates with its router and the weather API through HTTP. However, the format of the messages exchanged with the routers and with the API is different. The messages exchanged with the routers follow the representation described in Section 3.3.2. On the other hand, the weather API



```
❯ node CSA\ suite/client/client.js
Client listening on 8080
Attempting to authorize
Enter your password: password
Attempting to authorize
Enter your password: secretPassword
Quitting the protocol

 ~/De/B/routersMPST  development *1                    ✓ 17s
```

```
❯ PROTOCOL_PATH="./CSA Suite/client/CSA_client.json" node router.js
Router for c listening on port 8000
Implementing party c online.
Router for participant s not online
Router for participant a not online
Router for participant s online
Router for participant a online
Received message {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to dependency {"sender":"c","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"passwd"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"passwd"}
Received message {"sender":"c","receiver":"a","payload":"password"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"password"}
Received message {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to dependency {"sender":"c","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"passwd"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"passwd"}
Received message {"sender":"c","receiver":"a","payload":"secretPassword"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"secretPassword
"}
Received message {"sender":"s","receiver":"c","payload":"quit"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"quit"}
Forwarding message to dependency {"sender":"c","receiver":"a","payload":"quit"}
Received message {"sender":"c","receiver":"a","payload":"quit"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"quit"}
Protocol finalised. Shutting down router.

 ~/De/B/routersMPST  development *1                                        ✓ 10s
```

```
❯ node CSA\ suite/server/server.js
Server listening on 8081
Authorization denied!
Client authorized!

 ~/De/B/routersMPST  development *1                    ✓ 14s
```

```
❯ PROTOCOL_PATH="./CSA Suite/server/CSA_server.json" node router.js
Router for s listening on port 8001
Implementing party s online.
Router for participant c online
Router for participant a not online
Router for participant a online
Received message {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to dependency {"sender":"s","receiver":"a","payload":"login"}
Received message {"sender":"a","receiver":"s","payload":"auth"}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":"auth"}
Received message {"sender":"a","receiver":"s","payload":false}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":false}
Received message {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to dependency {"sender":"s","receiver":"a","payload":"login"}
Received message {"sender":"a","receiver":"s","payload":"auth"}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":"auth"}
Received message {"sender":"a","receiver":"s","payload":true}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":true}
Received message {"sender":"s","receiver":"c","payload":"quit"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"quit"}
Forwarding message to dependency {"sender":"s","receiver":"a","payload":"quit"}
Protocol finalised. Shutting down router.

 ~/De/B/routersMPST  development *1                                        ✓ 8s
```

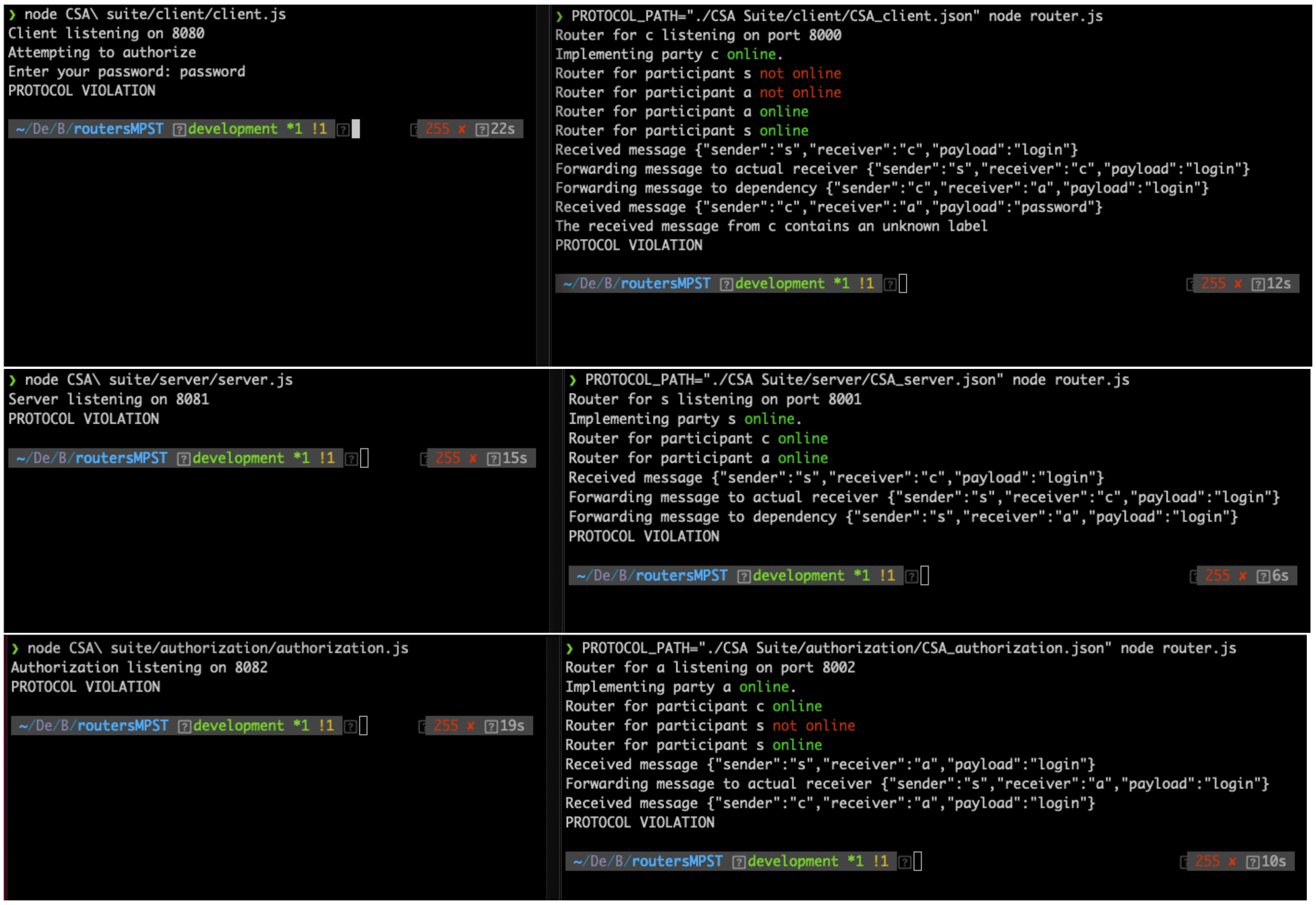
```
❯ node CSA\ suite/authorization/authorization.js
Authorization listening on 8082
Authorizing client
Verifying password
Authorizing client
Verifying password
Quitting the process

 ~/De/B/routersMPST  development *1                    ✓ 12s
```

```
Router for participant c online
Router for participant s online
Received message {"sender":"s","receiver":"a","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"passwd"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"passwd"}
Received message {"sender":"c","receiver":"a","payload":"password"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"password"}
Received message {"sender":"a","receiver":"s","payload":"auth"}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":"auth"}
Received message {"sender":"a","receiver":"s","payload":false}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":false}
Received message {"sender":"s","receiver":"a","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"passwd"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"passwd"}
Received message {"sender":"c","receiver":"a","payload":"secretPassword"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"secretPassword"
}
Received message {"sender":"a","receiver":"s","payload":"auth"}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":"auth"}
Received message {"sender":"a","receiver":"s","payload":true}
Forwarding message to actual receiver {"sender":"a","receiver":"s","payload":true}
Received message {"sender":"s","receiver":"a","payload":"quit"}
Forwarding message to actual receiver {"sender":"s","receiver":"a","payload":"quit"}
Received message {"sender":"c","receiver":"a","payload":"quit"}
Received message {"sender":"c","receiver":"a","payload":"quit"}
Forwarding message to actual receiver {"sender":"c","receiver":"a","payload":"quit"}
Protocol finalised. Shutting down router.

 ~/De/B/routersMPST  development *1                                        ✓ 6s
```

Figure 16: Execution of the Client-Server-Authorization protocol. All participant implementations adhere to the given protocol.



```
❯ node CSA\ suite/client/client.js
Client listening on 8080
Attempting to authorize
Enter your password: password
PROTOCOL VIOLATION

 ~/De/B/routersMPST ⍰development *1 !1 ⍰                    255 ✘ ⍰22s
```

```
❯ PROTOCOL_PATH="./CSA Suite/client/CSA_client.json" node router.js
Router for c listening on port 8000
Implementing party c online.
Router for participant s not online
Router for participant a not online
Router for participant a online
Router for participant s online
Received message {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to dependency {"sender":"c","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"password"}
The received message from c contains an unknown label
PROTOCOL VIOLATION

 ~/De/B/routersMPST ⍰development *1 !1 ⍰                    255 ✘ ⍰12s
```

```
❯ node CSA\ suite/server/server.js
Server listening on 8081
PROTOCOL VIOLATION

 ~/De/B/routersMPST ⍰development *1 !1 ⍰                    255 ✘ ⍰15s
```

```
❯ PROTOCOL_PATH="./CSA Suite/server/CSA_server.json" node router.js
Router for s listening on port 8001
Implementing party s online.
Router for participant c online
Router for participant a online
Received message {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"c","payload":"login"}
Forwarding message to dependency {"sender":"s","receiver":"a","payload":"login"}
PROTOCOL VIOLATION

 ~/De/B/routersMPST ⍰development *1 !1 ⍰                    255 ✘ ⍰6s
```

```
❯ node CSA\ suite/authorization/authorization.js
Authorization listening on 8082
PROTOCOL VIOLATION

 ~/De/B/routersMPST ⍰development *1 !1 ⍰                    255 ✘ ⍰19s
```

```
❯ PROTOCOL_PATH="./CSA Suite/authorization/CSA_authorization.json" node router.js
Router for a listening on port 8002
Implementing party a online.
Router for participant c online
Router for participant s not online
Router for participant s online
Received message {"sender":"s","receiver":"a","payload":"login"}
Forwarding message to actual receiver {"sender":"s","receiver":"a","payload":"login"}
Received message {"sender":"c","receiver":"a","payload":"login"}
PROTOCOL VIOLATION

 ~/De/B/routersMPST ⍰development *1 !1 ⍰                    255 ✘ ⍰10s
```

Figure 17: Execution of the Client-Server-Authorization protocol. The client implementation does not adhere to the given protocol.

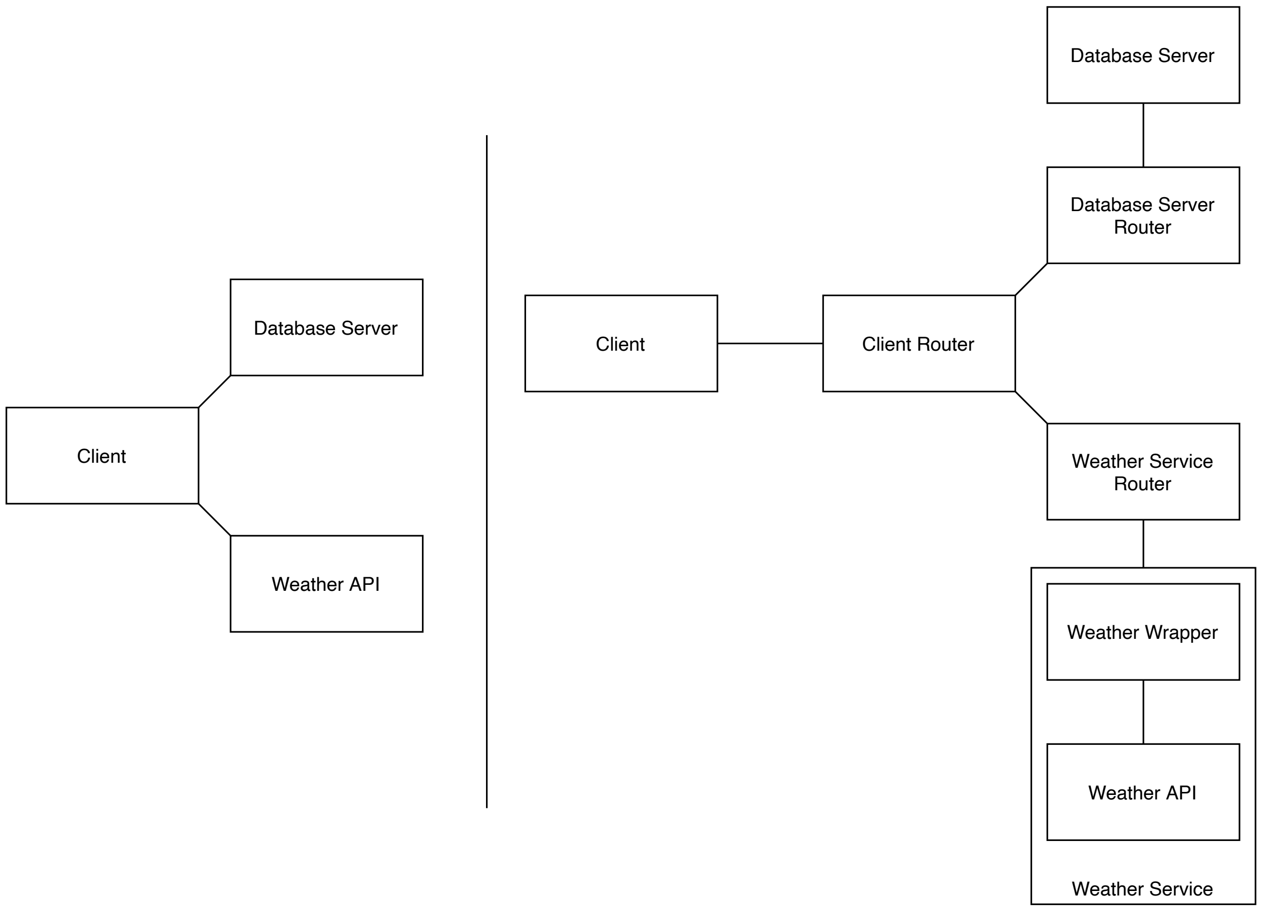


Figure 18: Network of participants before and after the routers and the weather wrapper were deployed.

can be used by making GET requests on the URL `https://api.openweathermap.org/data/2.5/weather`. Compared to the messages exchanged with the routers, the wrapper encodes all the essential information needed by the weather API in its URL, such as the latitude, longitude, preferred unit system and the API key.

The weather API on its own cannot handle the dependency labels that are transmitted in the network, nor does it deal with protocol violations. Nonetheless, the weather wrapper is able to take dependency labels into account by encoding them as states in its finite state machine. Moreover, the weather wrapper handles POST requests on the route `/api/violation/`, such that it can react accordingly when its router signals that a protocol violation has occurred. It is important to note that from the perspective of the routers and the other participants, the weather wrapper and the weather API act as a whole participant. Thus, if the weather wrapper is notified of a protocol violation, it terminates its execution. Subsequently, further communication with the API becomes impossible, even if the API itself is still online and working. Furthermore, if the weather API fails or if it becomes unavailable, the weather wrapper terminates accordingly.

Figure 19 shows the execution of the protocol when all participants adhere to the given protocol, whereas Figure 20 displays the execution of the protocol when the weather service produces a protocol violation by sending the temperature as a string value instead of a real value. The first column of terminal windows displays the output of the client, database, and weather service implementations, and the second column of terminal windows displays the output of the client, database and weather service router processes. The client requests the weather in three different cities: `Amsterdam`, `Groningen`, and `Gronningen`. The first two queries yield the temperatures in those corresponding cities, whereas the third query yields an error, since there is no city `Gronningen` in the database. At last, the client quits the application. The second scenario is meant to highlight that even if the underlying weather API suffers any changes, the library is able to detect protocol violations.

6 Transpiling global types to JSON

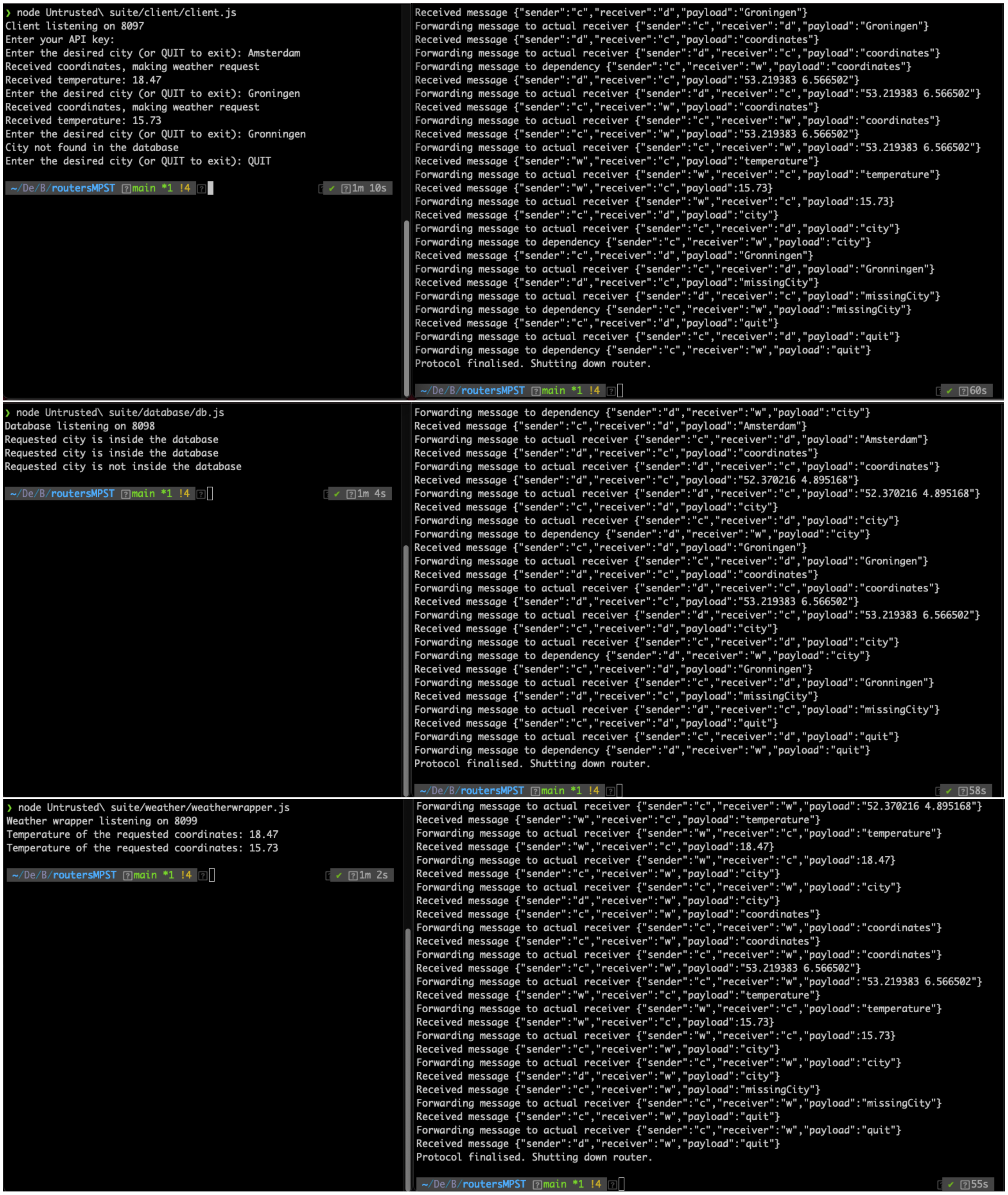
The protocol specification interface uses JSON to encode global types. JSON offers a clear key-value format, and it follows a tree structure. Nevertheless, protocols that involve numerous choices or long message sequences can be cumbersome to represent in JSON. To ease the encoding of protocols in JSON, we have augmented the project with a transpiler: a program that takes a global type written in some language and converts it to an equivalent global type in a different language.

This section describes the extensions that were added on top of the original project. It describes a new specification language that is used to represent global types. Thereafter, this section discusses the implementation of a transpiler and the steps involved in translating global types given in the GT specification language to JSON. For the development of the transpiler, we use Rascal [11, 12]. Rascal is a meta-programming language that is used to develop several tools for domain specific languages, as it provides lots of features relevant for software language engineering.

The transpiler builds on top of a a work-in-progress compiler built by Van den Heuvel. It converts global types written in the GT (global type) specification language to their JSON equivalent. The GT specification language offers a syntax similar to the formal notation of global types in the literature. Hence, it is much easier to write and comprehend. The following grammar enumerates the syntax of the GT specification language using Backus-Naur Form.

$$\begin{array}{lcl}
\langle \textit{globalType} \rangle & ::= & \langle \textit{participant} \rangle \text{ `\texttt{->}' } \langle \textit{participant} \rangle \text{ `\texttt{<}' } \langle \textit{valueType} \rangle \text{ `\texttt{>}' `\texttt{.}' } \langle \textit{globalType} \rangle \\
 & | & \langle \textit{participant} \rangle \text{ `\texttt{->}' } \langle \textit{participant} \rangle \text{ `\texttt{(}' } \langle \textit{choices} \rangle \text{ `\texttt{)}'} \\
 & | & \text{`\texttt{mu}' } \langle \textit{var} \rangle \text{ `\texttt{.}' } \langle \textit{globalType} \rangle \\
 & | & \langle \textit{var} \rangle \\
 & | & \text{`\texttt{end}'}
\end{array}$$

$$\begin{array}{lcl}
\langle \textit{choices} \rangle & ::= & \langle \textit{choices} \rangle \text{ `\texttt{,}' } \langle \textit{label} \rangle \text{ `\texttt{.}' } \langle \textit{globalType} \rangle \\
 & | & \langle \textit{label} \rangle \text{ `\texttt{.}' } \langle \textit{globalType} \rangle
\end{array}$$



```
❯ node Untrusted\ suite/client/client.js
Client listening on 8097
Enter your API key:
Enter the desired city (or QUIT to exit): Amsterdam
Received coordinates, making weather request
Received temperature: 18.47
Enter the desired city (or QUIT to exit): Groningen
Received coordinates, making weather request
Received temperature: 15.73
Enter the desired city (or QUIT to exit): Gronningen
City not found in the database
Enter the desired city (or QUIT to exit): QUIT

~/De/B/routersMPST main *1 !4                                    ✓ 1m 10s
```

```
Received message {"sender":"c","receiver":"d","payload":"Groningen"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"Groningen"}
Received message {"sender":"d","receiver":"c","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"coordinates"}
Forwarding message to dependency {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"d","receiver":"c","payload":"53.219383 6.566502"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"53.219383 6.566502"}
Received message {"sender":"c","receiver":"w","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"c","receiver":"w","payload":"53.219383 6.566502"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"53.219383 6.566502"}
Received message {"sender":"w","receiver":"c","payload":"temperature"}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":"temperature"}
Received message {"sender":"w","receiver":"c","payload":15.73}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":15.73}
Received message {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to dependency {"sender":"c","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"d","payload":"Gronningen"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"Gronningen"}
Received message {"sender":"d","receiver":"c","payload":"missingCity"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"missingCity"}
Forwarding message to dependency {"sender":"c","receiver":"w","payload":"missingCity"}
Received message {"sender":"c","receiver":"d","payload":"quit"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"quit"}
Forwarding message to dependency {"sender":"c","receiver":"w","payload":"quit"}
Protocol finalised. Shutting down router.

~/De/B/routersMPST main *1 !4                                                            ✓ 60s
```

```
❯ node Untrusted\ suite/database/db.js
Database listening on 8098
Requested city is inside the database
Requested city is inside the database
Requested city is not inside the database

~/De/B/routersMPST main *1 !4                                    ✓ 1m 4s
```

```
Forwarding message to dependency {"sender":"d","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"d","payload":"Amsterdam"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"Amsterdam"}
Received message {"sender":"d","receiver":"c","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"coordinates"}
Received message {"sender":"d","receiver":"c","payload":"52.370216 4.895168"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"52.370216 4.895168"}
Received message {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to dependency {"sender":"d","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"d","payload":"Groningen"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"Groningen"}
Received message {"sender":"d","receiver":"c","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"coordinates"}
Received message {"sender":"d","receiver":"c","payload":"53.219383 6.566502"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"53.219383 6.566502"}
Received message {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to dependency {"sender":"d","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"d","payload":"Gronningen"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"Gronningen"}
Received message {"sender":"d","receiver":"c","payload":"missingCity"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"missingCity"}
Received message {"sender":"c","receiver":"d","payload":"quit"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"quit"}
Forwarding message to dependency {"sender":"d","receiver":"w","payload":"quit"}
Protocol finalised. Shutting down router.

~/De/B/routersMPST main *1 !4                                                            ✓ 58s
```

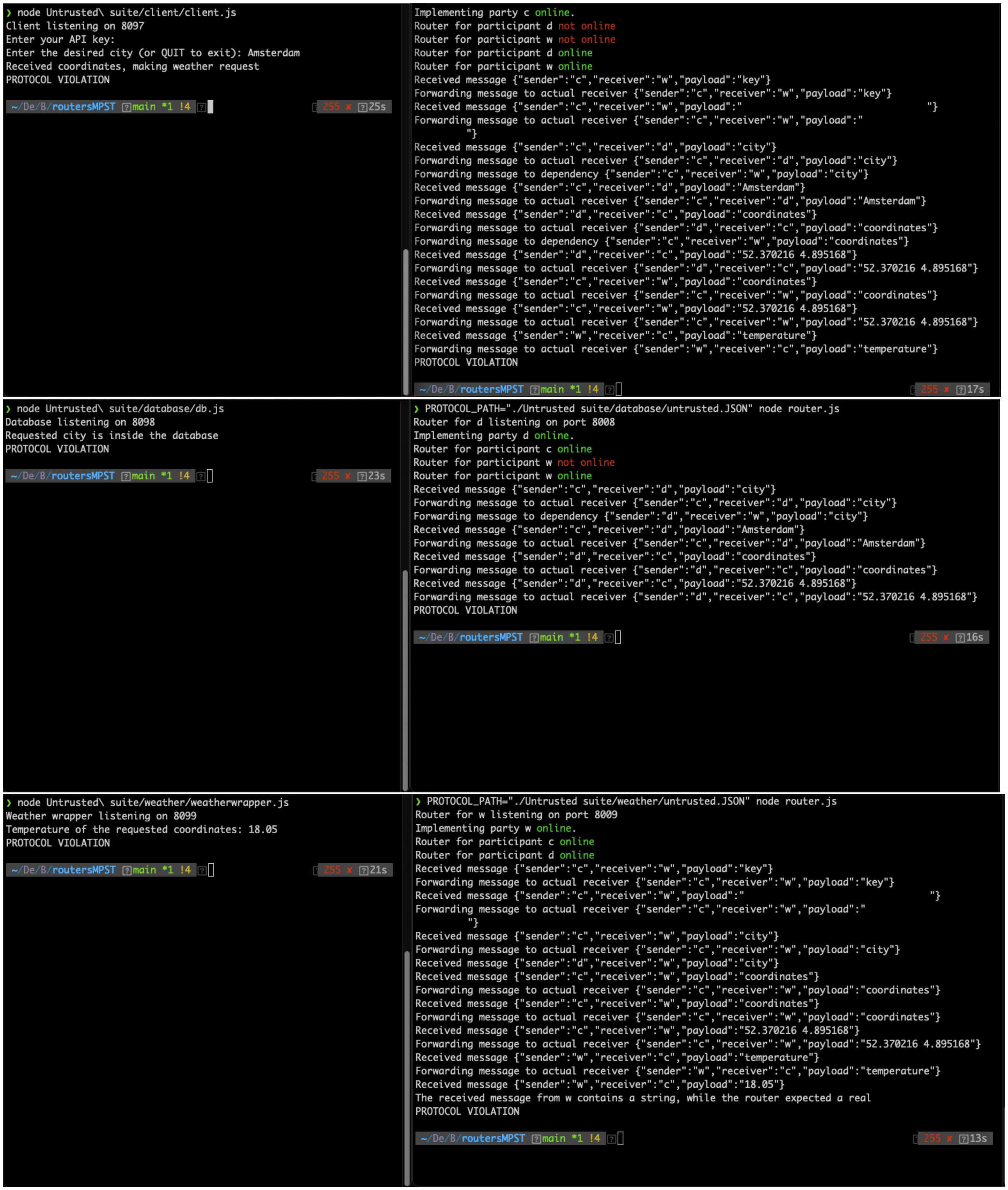
```
❯ node Untrusted\ suite/weather/weatherwrapper.js
Weather wrapper listening on 8099
Temperature of the requested coordinates: 18.47
Temperature of the requested coordinates: 15.73

~/De/B/routersMPST main *1 !4                                    ✓ 1m 2s
```

```
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"52.370216 4.895168"}
Received message {"sender":"w","receiver":"c","payload":"temperature"}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":"temperature"}
Received message {"sender":"w","receiver":"c","payload":18.47}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":18.47}
Received message {"sender":"c","receiver":"w","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"city"}
Received message {"sender":"d","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"w","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"c","receiver":"w","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"c","receiver":"w","payload":"53.219383 6.566502"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"53.219383 6.566502"}
Received message {"sender":"w","receiver":"c","payload":"temperature"}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":"temperature"}
Received message {"sender":"w","receiver":"c","payload":15.73}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":15.73}
Received message {"sender":"c","receiver":"w","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"city"}
Received message {"sender":"d","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"w","payload":"missingCity"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"missingCity"}
Received message {"sender":"c","receiver":"w","payload":"quit"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"quit"}
Received message {"sender":"d","receiver":"w","payload":"quit"}
Protocol finalised. Shutting down router.

~/De/B/routersMPST main *1 !4                                                            ✓ 55s
```

Figure 19: Execution of the G_{weather} protocol when all participants adhere to it.



```
❯ node Untrusted\ suite/client/client.js
Client listening on 8097
Enter your API key:
Enter the desired city (or QUIT to exit): Amsterdam
Received coordinates, making weather request
PROTOCOL VIOLATION

 ~/De/B/routersMPST main *1 !4                                 255 x 25s
```

```
Implementing party c online.
Router for participant d not online
Router for participant w not online
Router for participant d online
Router for participant w online
Received message {"sender":"c","receiver":"w","payload":"key"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"key"}
Received message {"sender":"c","receiver":"w","payload":"                                "}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"
         "}
Received message {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to dependency {"sender":"c","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"d","payload":"Amsterdam"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"Amsterdam"}
Received message {"sender":"d","receiver":"c","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"coordinates"}
Forwarding message to dependency {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"d","receiver":"c","payload":"52.370216 4.895168"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"52.370216 4.895168"}
Received message {"sender":"c","receiver":"w","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"c","receiver":"w","payload":"52.370216 4.895168"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"52.370216 4.895168"}
Received message {"sender":"w","receiver":"c","payload":"temperature"}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":"temperature"}
PROTOCOL VIOLATION

 ~/De/B/routersMPST main *1 !4                                                     255 x 17s
```

```
❯ node Untrusted\ suite/database/db.js
Database listening on 8098
Requested city is inside the database
PROTOCOL VIOLATION

 ~/De/B/routersMPST main *1 !4                                 255 x 23s
```

```
❯ PROTOCOL_PATH="./Untrusted suite/database/untrusted.JSON" node router.js
Router for d listening on port 8008
Implementing party d online.
Router for participant c online
Router for participant w not online
Router for participant w online
Received message {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"city"}
Forwarding message to dependency {"sender":"d","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"d","payload":"Amsterdam"}
Forwarding message to actual receiver {"sender":"c","receiver":"d","payload":"Amsterdam"}
Received message {"sender":"d","receiver":"c","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"coordinates"}
Received message {"sender":"d","receiver":"c","payload":"52.370216 4.895168"}
Forwarding message to actual receiver {"sender":"d","receiver":"c","payload":"52.370216 4.895168"}
PROTOCOL VIOLATION

 ~/De/B/routersMPST main *1 !4                                                     255 x 16s
```

```
❯ node Untrusted\ suite/weather/weatherwrapper.js
Weather wrapper listening on 8099
Temperature of the requested coordinates: 18.05
PROTOCOL VIOLATION

 ~/De/B/routersMPST main *1 !4                                 255 x 21s
```

```
❯ PROTOCOL_PATH="./Untrusted suite/weather/untrusted.JSON" node router.js
Router for w listening on port 8009
Implementing party w online.
Router for participant c online
Router for participant d online
Received message {"sender":"c","receiver":"w","payload":"key"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"key"}
Received message {"sender":"c","receiver":"w","payload":"                                "}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"
         "}
Received message {"sender":"c","receiver":"w","payload":"city"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"city"}
Received message {"sender":"d","receiver":"w","payload":"city"}
Received message {"sender":"c","receiver":"w","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"c","receiver":"w","payload":"coordinates"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"coordinates"}
Received message {"sender":"c","receiver":"w","payload":"52.370216 4.895168"}
Forwarding message to actual receiver {"sender":"c","receiver":"w","payload":"52.370216 4.895168"}
Received message {"sender":"w","receiver":"c","payload":"temperature"}
Forwarding message to actual receiver {"sender":"w","receiver":"c","payload":"temperature"}
Received message {"sender":"w","receiver":"c","payload":"18.05"}
The received message from w contains a string, while the router expected a real
PROTOCOL VIOLATION

 ~/De/B/routersMPST main *1 !4                                                     255 x 13s
```

Figure 20: Execution of the G_{weather} protocol when the weather service produces a violation.

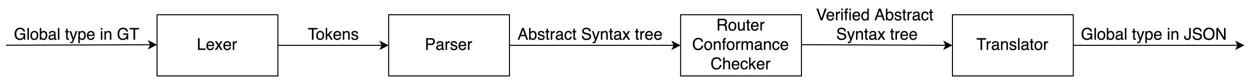


Figure 21: Overview of the transpiler pipeline.

$\langle participant \rangle$ ::= [a-zA-Z_][a-zA-Z0-9_]*

$\langle valueType \rangle$ ::= [a-zA-Z_][a-zA-Z0-9_]*

$\langle var \rangle$ ::= [A-Z][0-9]*

$\langle label \rangle$ ::= [a-zA-Z0-9]+

The grammar of the GT specification language starts with the *globalType* symbol. It can either be substituted with an exchange between two participants involving a value of some type, an exchange between two participants involving labelled choices, a recursive definition, a recursive call, or the end of the protocol. The choices of an exchange can either be a list of labels and continuations delimited by commas, or a single label and continuation. A participant is represented by a string consisting of a lowercase letter, an uppercase letter, or an underscore, followed by zero or more lowercase or uppercase letters, underscores or digits. The value type also consists of a string following the same rules as the participant. Recursion variables are simply strings consisting of a capital letter and zero or more digits. Each label is a string consisting of one or more lowercase or uppercase letters, or digits. Similar to the syntax of global types, sequencing is denoted through '.'. The GT specification language supports single-line comments as well, denoted through `//`. To put matters into perspective, we can write the global type of the Client-Server-Authorization protocol enumerated in Equation (2) using the GT specification language as follows.

```
mu X . s -> c (login . c -> a (passwd . c -> a <str> . a -> s (auth. a -> s <bool> . X)),
               quit . c -> a (quit . end))
```

The transpiler consists of four main operations: lexing, parsing, router conformance checking, and translating. Together, they form a pipeline, and each component offers a different representation of the global type to the next component. Figure 21 displays the transpiler pipeline and the inputs/outputs of each operation.

The lexer reads the global type given in the GT specification language and outputs tokens. Each token is defined through a regular expression, and it identifies a terminal symbol in the given grammar. Each regular expression is represented through a finite state machine. The lexer then combines all finite state machines into a single finite state machine, and it runs the global type text through it. To check whether a string can be generated from a regular expression, the string is run through the finite state machine. For each character of the string, the finite state machine tries to take an appropriate transition to a new state. If no such transition exists, the string cannot be generated from the regular expression. Once all characters from the string have been used, the finite state machine checks whether it reached an accepting state. If the finite state machine is in an accepting state, the string can be generated from the regular expression. Hence, a token is provided to the next component in the pipeline, the parser. Otherwise, the given string cannot be generated using the regular expression.

The parser uses the collection of tokens provided by the lexer to check whether they can be generated from the given grammar. To this end, the parser provided by Rascal proceeds in a top-down approach: it starts with the symbol *globalType*, and it applies production rules recursively until it generates the collection of tokens provided by the lexer. If the tokens provided by the lexer cannot be reproduced by the parser, a syntax error is generated. Otherwise, the parser returns a parse tree.

Rascal offers methods for reading, lexing and parsing a file, so the user simply needs to define the syntax rules and the regular expressions corresponding to each token. Hence, the lexer and the parser are usually combined into a single component. The parse tree returned by the parser contains information about each token, such as its location in the source text, its lexeme, whitespace, comments and more. However, most of them are irrelevant for the translation procedure. Thus, the transpiler converts the parse tree to an abstract syntax tree (AST) to abstract away all the unnecessary details through an implosion procedure provided by Rascal. The abstract syntax tree is built according to an abstract syntax. The abstract syntax is similar to the concrete syntax, but it does not specify regular expressions, whitespace rules or comments.

The GT specification language was originally built to compile to a different language than JSON. Thus, the global types for have a slightly different syntax than what we expect for our toolkit. The GT specification language allows two participants to exchange a value without exchanging any label whatsoever. Moreover, the GT specification language allows users to define their own types. Hence, valid GT specifications might not be entirely compatible with our library. To tackle this issue, the transpiler employs a router conformance check operation, which constraints value exchanges to be preceded by a corresponding label, and restricts value types to integers, real numbers, strings and booleans.

The algorithm that performs the router conformance check traverses the abstract syntax tree recursively, inspecting the type of the first interaction at each step. Upon completion, it returns a list of conformance errors that make the current global type incompatible with our tool. If the list is empty, the user can proceed with the translation procedure.

- If the procedure encounters a value exchange between two participants, it generates a conformance error, as it is not preceded by a labelled exchange. Next, it checks the value type for additional conformance errors, and then it checks the continuation of the global type.
- If the procedure encounters a labelled exchange, the continuation of each branch is verified:
  - If the branch's continuation is a value exchange, the procedure checks whether the sender and the receiver of the value coincide with the sender and the receiver of the label. If the pairs are different, the algorithm generates a conformance error. Then, it checks the value type, and it applies the procedure to the continuation of the exchange.
  - Otherwise, the algorithm is applied recursively to the branch's continuation.
- If the procedure encounters a recursive definition, the algorithm is applied to the protocol continuation.
- If the procedure encounters a recursive call or the end of the protocol, the procedure ends.

To check the value type, the algorithm simply checks if it is an integer (*int*), a real number (*real*), a boolean (*bool*), or a string (*str*). Any other type generates a conformance error, as it is not compatible with the library.

The translation procedure uses string templates to generate the equivalent JSON encoding. It traverses the abstract syntax tree recursively, and it generates the JSON equivalent at each step. It performs a case-by-case analysis on the type of the first interaction in the global type. Each type of interaction has a predefined string containing JSON syntax following the convention described in Section 3.3.1. Thereafter, the algorithm replaces the values in those predefined strings to generate the appropriate JSON text.

7 Related and future work

This section discusses other practical applications of the multiparty session types framework. It first dives into static verification tools, their advantages and disadvantages. Thereafter, other tools employing multiparty session types dynamically are described, followed by a discussion of possible future extensions to the current project.

7.1 Static verification

The theory of session types is often employed statically to assess the conformance of multiple participants to a given protocol by checking the implementation of each participant. While this method permits a thorough verification of a participant implementation, it cannot always be employed. In a distributed system, components are usually implemented in a language that best suits their individual needs. Hence, the heterogeneity of a distributed system often impedes the applicability of static verification tools, as it is difficult to design an algorithm to verify an implementation independently of its source language. Furthermore, it is rarely the case that all participant implementations are available and accessible. There are several static implementations employing multiparty session types to verify protocol conformance, such as [3, 4, 9, 13, 15].

Honda et al. [8] introduce Scribble, a protocol specification language. Scribble relies on multiparty session types to allow users to define protocols, and it uses the projection of the global type onto each participant to obtain local types. Scribble employs local types and a Conversation API to facilitate the static verification of participant implementations. Ng and Yoshida [19] develop Pabble, an extension of Scribble, that increases its expressivity by using parameterised types. Parameterised types and protocols permit an efficient declaration of collections of participants.

Kouzapas et al. [14] develop a tool to statically analyse the protocol conformance of communicating participants in Java. Their tool, Mungo, allows users to augment their classes with typestates. Each class coupled with a typestate has a finite state machine defining the behaviour of their objects, in terms of what methods can be executed, and their corresponding order. During compilation, Mungo checks the objects of each class coupled with a typestate to see whether the correct methods are invoked, and their order is respected. In addition to Mungo, Kouzapasa et al. create StMungo, a tool that is able to generate typestates equivalent to protocols specified in Scribble. The protocol is first specified in Scribble, projected onto each participant, and then StMungo creates typestates from the previously obtained local types.

Neykova et al. [17] develop a tool in F# capable of performing protocol conformance verification during compilation. Their tool relies on Scribble for the specification of the protocol, and it works by employing type providers to check the implementations of the participants through the perspective of multiparty session types. Furthermore, their tool is able to handle refinements in the form of additional constraints imposed on the interactions between communication participants.

King et al. [10] take a slightly different approach to static verification methods. They implement a tool which generates source code according to the specification of the protocol in PureScript, a language that follows the functional programming paradigm and compiles to JavaScript. In their work, the global type is specified using Scribble, which is used to generate finite state machines identifying the interactions of a participant. The states of the finite state machines are abstracted as data types, and they are used to steer the communication of the underlying application.

Ng et al. [20] develop a framework to check the conformance of communicating participants to a given protocol in C. Their tool also relies on Scribble for the specification of the protocol and for the local types identifying specific participants. Their framework includes a component that handles the verification of a participant implementation against a local type, and a component that facilitates the communication with other participants.

7.2 Dynamic verification

Tools relying on dynamic verification methods benefit from a wide range of advantages, such as verifying protocol conformance independently of the underlying participant implementations or their corresponding source languages. Nonetheless, they can only observe and verify the interactions that occur between participants. Hence, dynamic verification methods are limited in the sense that they cannot analyse the exchanges that do not take place.

Neykova et al. [18] develop a tool to monitor protocol conformance at runtime in Python, SPY. SPY relies on Scribble for the specification of the protocol and local projection. From the local types, SPY creates finite state machines, which are used by monitors to verify protocol conformance. Similar to our tool, SPY uses monitors distributed at each participant to verify protocol conformance. Additionally, SPY offers two types of monitors, internal and external, and it is meant to monitor protocol conformance for participant implementations that use Python as their source language. On the other hand, our tool is able to monitor protocol conformance independently of the source language of the participant implementations.

Building on top of multiparty session types, Neykova et al. [16] implement a tool that verifies protocol conformance at runtime in Python, but the type system that was employed facilitates the use of various time constraints. Their tool also relies on Scribble for the specification of the protocol.

Burlò et al [1] create a tool to monitor conformance for binary protocols in Scala. Their tool targets probabilistic session types, which enhance the expressivity of session types with probabilities. Their tool uses only one monitor for two participants, as opposed to our tool, which uses a monitor for each interacting participant.

7.3 Future work

Our tool offers lots of possibilities for future extensions. Our tool currently restricts the set of available types to integers, real numbers, strings, and booleans. Thus, we would like to extend the current set of available types, and we would also like to allow the user to define custom types using certain primitives. Additionally, we would like to implement an authentication mechanism for the routers and the participants of the communication. In their current state, routers are vulnerable to attacks from malicious participants pretending to be someone else. Hence, an authentication procedure to guarantee the authenticity of the received messages and the senders would mitigate some of the security risks involved.

8 Conclusion

We have implemented a tool in JavaScript to verify protocol conformance dynamically in a network of communicating participants. The tool is able to observe the messages exchanged in a given network and analyse them internally to check whether they conform to the given protocol. It generates new components, routers, for each participant. Each router oversees the messages exchanged with its corresponding participant, it analyses them to see whether they conform to the given protocol, and it reports protocol violations if the observed messages deviate from the protocol. In doing so, protocol conformance is verified in a distributed, decentralised manner.

Routers employ multiparty session types to determine the messages that must be exchanged between certain participants. More specifically, they use relative types and the relative projection operation defined by Van den Heuvel and Pérez [23] to extract the types of the messages exchanged between pairs of participants. Furthermore, routers use the router synthesis algorithm defined by Van den Heuvel and Pérez [23], and they encode the actions a router must perform and the specifics of each message as finite state machines. Subsequently, finite state machines define the behaviour of a particular router, and they steer the communication in the network. Hence, our tool can be regarded as a faithful dynamic implementation of the formal model defined by Van den Heuvel and Pérez [23], as routers rely on the relative type theory and they closely follow the router synthesis algorithm to determine what messages must be exchanged in a distributed system.

We deploy the implemented library in two different scenarios to showcase its properties. We implement the Client-Server-Authorization protocol by Van den Heuvel and Pérez [23], adapted from [22], to illustrate the application of the tool when all participants adhere to the given protocol, and when a participant produces a protocol violation. Thereafter, we highlight the compatibility of the aforementioned tool with any network of communicating participants, by deploying it in a scenario where not all participant implementations are compatible with routers.

At last, we develop a transpiler to aid the specification of the protocol in JSON. We build on top of a global type specification language by Bas van den Heuvel in Rascal, and we augment it to support translation to our JSON specification. Consequently, the Rascal lexer and parser are extended with a router conformance checker to ensure the compatibility of the global type with the JSON format used by the tool.

Acknowledgements

I would like to thank my supervisors, Bas van den Heuvel and Jorge A. Pérez, for their continuous support throughout the entirety of the bachelor's project, numerous feedback sessions and weekly meetings. At last, I would like to thank my family for supporting me throughout the completion of the project.

References

[1] Christian Bartolo Burlò, Adrian Francalanza, Alceste Scalas, Catia Trubiani, and Emilio Tuosto. Towards probabilistic session-type monitoring. *CoRR*, abs/2107.08729, 2021. doi: https://doi.org/10.48550/arXiv.2107.08729.

[2] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. volume 7173, pages 25–45, 06 2011. doi: https://doi.org/10.1007/978-3-642-30065-3_2.

[3] Zak Cutner and Nobuko Yoshida. Safe session-based asynchronous coordination in rust. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages*, pages 80–89, Cham, 2021. Springer International Publishing. doi: https://doi.org/10.1007/978-3-030-78142-2_5.

[4] Folkert de Vries and Jorge A. Pérez. Reversible session-based concurrency in haskell. In Michał Pałka and Magnus Myreen, editors, *Trends in Functional Programming*, pages 20–45, Cham, 2019. Springer International Publishing. doi: https://doi.org/10.1007/978-3-030-18506-0_2.

[5] OpenJS Foundation. Express. url: https://expressjs.com/ (access date: 1st of June, 2022).

[6] OpenJS Foundation. Node.js. url: https://nodejs.org/ (access date: 1st of June, 2022).

[7] Simon Gay, Vasco Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Caldeira. Modular session types for distributed object-oriented programming. volume 45, pages 299–312, 01 2010. doi: https://doi.org/10.1145/1707801.1706335.

[8] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. volume 6536, pages 55–75, 02 2011. doi: https://doi.org/10.1007/978-3-642-19056-8_4.

[9] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, page 13–22, New York, NY, USA, 2015. Association for Computing Machinery. doi: https://doi.org/10.1145/2808098.2808100.

[10] Jonathan King, Nicholas Ng, and Nobuko Yoshida. Multiparty session type-safe web development with static linearity. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 35–46, 2019. doi: https://doi.org/10.4204/EPTCS.291.4.

[11] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, 2009. doi: https://doi.org/10.1109/SCAM.2009.28.

[12] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal, 10 years later. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 139–139, 2019. doi: https://doi.org/10.1109/SCAM.2019.00023.

[13] Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, page 1–13, New York, NY, USA, 2021. Association for Computing Machinery. doi: https://doi.org/10.1145/3471874.3472979.

[14] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Science of Computer Programming*, 155:52–75, 2018. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016, doi: https://doi.org/10.1016/j.scico.2017.10.006.

[15] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Implementing multiparty session types in rust. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages*, pages 127–136, Cham, 2020. Springer International Publishing. doi: https://doi.org/10.1007/978-3-030-50029-0_8.

[16] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Form. Asp. Comput.*, 29(5):877–910, sep 2017. doi: https://doi.org/10.1007/s00165-017-0420-8.

[17] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: Compile-time api generation of distributed protocols with refinements in f#. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 128–138, New York, NY, USA, 2018. Association for Computing Machinery. doi: https://doi.org/10.1145/3178372.3179495.

[18] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local verification of global protocols. pages 358–363, 09 2013. doi: https://doi.org/10.1007/978-3-642-40787-1_25.

[19] Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised scribble for parallel programming. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 707–714, 2014. doi: https://doi.org/10.1109/PDP.2014.20.

[20] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session c: Safe parallel programming with message optimisation. volume 7304, pages 202–218, 05 2012. doi: https://doi.org/10.1007/978-3-642-30561-0_15.

[21] OpenWeather. One call weather api. url: https://openweathermap.org/api/one-call-3 (access date: 14th of June, 2022).

[22] Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: https://doi.org/10.1145/3290343.

[23] Bas van den Heuvel and Jorge A. Pérez. A decentralized analysis of multiparty protocols. *Science of Computer Programming*, page 102840, 2022. doi: https://doi.org/10.1016/j.scico.2022.102840.

[24] Nobuko Yoshida and Lorenzo Gheri. A Very Gentle Introduction to Multiparty Session Types. In *16th International Conference on Distributed Computing and Internet Technology*, volume 11969 of *LNCS*, pages 73–93. Springer, 2020. doi: https://doi.org/10.1007/978-3-030-36987-3_5.

[25] Xinchang Zhang, Meihong Yang, Guanggang Geng, and Wanming Luo. A dfsm-based protocol conformance testing and diagnosing method. *Informatica*, 22(3):447–469, jul 2011. doi: https://doi.org/10.15388/Informatica.2011.336.