Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types

Anonymous Author(s)

Abstract

Modern web programming involves coordinating interactions between web browser clients and a web server. Typically, the interactions in web-based distributed systems are informally described, making it difficult to ensure communication correctness, or *communication safety*, i.e. all endpoints progress their communications without type errors or deadlocks, conforming to a given multiparty protocol.

We present STScript, a toolchain that generates TypeScript APIs for communication-safe web development over WebSockets, and RouST, a new session type theory that supports multiparty communications with routing mechanisms.

STScript provides developers with TypeScript APIs generated from a communication protocol specification based on RouST. The generated APIs build upon TypeScript concurrency practices, complement the event-driven style of programming in full-stack web development, and are compatible with the Node.js runtime for server-side endpoints and the React.js framework for browser-side endpoints.

RouST can express multiparty interactions routed via an intermediate participant. It supports peer-to-peer communication between browser-side endpoints by routing communication via the server in a way that avoids excessive serialisation. RouST guarantees communication safety for endpoint web applications written using STScript APIs.

We evaluate the expressiveness of STScript for modern web programming using several production-ready case studies deployed as web applications.

Keywords: TypeScript, WebSocket, API generation, session types, deadlock freedom

1 Introduction

Web technology advancements have changed the way people use computers. Many services that required standalone applications, such as email, chat, video conferences, or even games, are now provided in a browser. While the Hypertext Transfer Protocol (HTTP) is widely used for serving web pages, its Request-Response model limits the communication patterns — the server may not send data to a client without the client first making a request.

The *WebSocket protocol* [12] addresses this limitation by providing a bi-directional channel between the client and the server, akin to a Unix socket. Managing the correct usage of WebSockets introduces an additional concern in the development process, due to a lack of WebSocket testing tools,[1] requiring an (often ad-hoc) specification of the communication protocol between server and clients.

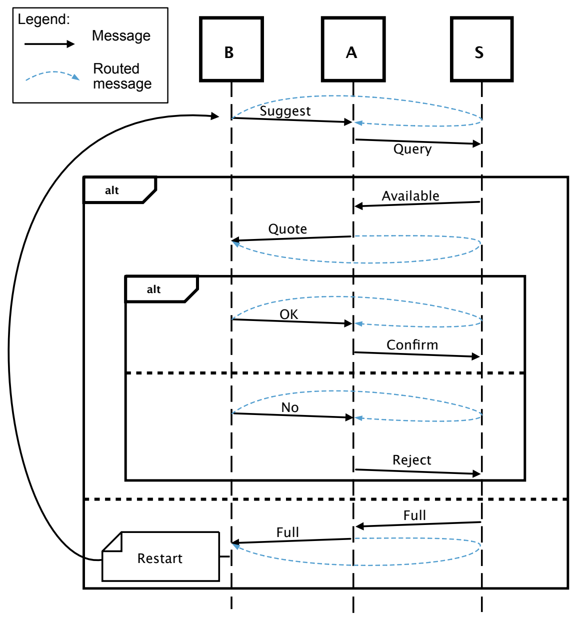

Figure 1. Travel Agency Protocol as a Sequence Diagram

Consider the scenario in Fig. 1, where an online travel agency operates a “travelling with a friend” scheme. It starts when a traveller (**B**) suggests a trip destination to their friend (**A**), who then queries the travel agency (**S**) if the trip is available. If so, the friends discuss among themselves whether to accept or reject the quoted price. If the trip was unavailable, the friends start again with a new destination.

An implementation of the travel agency protocol may contain programming errors, risking *communication safety*. For example, the following implementation of the client-side endpoint for traveller **A** sending a quote to traveller **B**.

```
<input type='number' id='quote' />
<button id='submitQuote'>Send Quote to B</button>
<script>
document.getElementById('submitQuote')
  .addEventListener('click', () => {
    const quote = document.getElementById('quote').value;
    travellerB.send({ label: 'quote', quote });
    travellerB.onMessage( /* go to different screen */ );
    /* ...snip... */ }); </script>
```

[1] Servers and clients tested separately using e.g. https://github.com/lensesio/cypress-websocket-testing/ and https://www.websocket.org/echo.html

PL'18, January 01–03, 2018, New York, NY, USA
2018.

There are subtle errors that violate the communication protocol, but these bugs are unfortunately left for the developer to manually identify and test against:

Communication Mismatch Whilst the input field mandates a *numerical value* (Line 1) for the quote, the `value` from the input field is actually a `string`. If **B** expects a `number` and performs arithmetic operations on the received payload from **A**, the type mismatch may be left hidden due to implicit type coercion and cause unintended errors.

Channel Usage Violation As **B** may take time to respond, **A** can experience a delay between sending the quote and receiving a response. Notice that the button remains *active* after sending the quote — **A** could click on the button again, and send additional quotes (thus reusing the communication channel), but **B** may be unable to deal with extra messages.

Handling Session Cancellation An additional concern is how to handle browser disconnections, as both travellers can freely close their browsers at any stage of the protocol. Suppose **S** temporarily reserves a seat on **A**'s query. If **A** closes their browser, the developer would need to make sure that **A** notifies **S** prior to disconnecting, and **S** needs to implement recovery logic (e.g. releasing the reserved seat) accordingly.

To prevent these errors and ensure deadlock-freedom, we propose to apply *session types* [14, 15] into practical interactive web programming. The scenario described in Fig. 1 can be precisely described with a *global type* using the typing discipline of *multiparty session types* (MPST) [15]. Well-typed implementations conform to the given *global protocol*, are guaranteed free from communication errors *by construction*.

Whereas session type programming is well-studied [1], its application on web programming, in particular, interactive web applications, remains relatively unexplored. Integrating session types with web programming has been piloted by recent work [13, 20, 22], yet none is able to seamlessly implement the previous application scenario: Fowler [13] uses *binary* (2-party) session types; and King et al. [20] require each non-server role to only communicate to the server, hence preventing interactions between non-server roles (cf. talking to a friend in the scenario). The programming languages used in these works are, respectively, Links [8] and PureScript [26], both not usually considered mainstream in the context of modern web programming. The Jolie language [22] focuses more on the server side, with limited support for an interactive front end of web applications.

This paper presents a toolchain, *Session TypeScript* (STScript), for implementing multiparty protocols safely in web programming. STScript integrates with *modern* tools and practices, utilising the popular programming language TypeScript, front end framework React.js and back end runtime Node.js. Developers first specify a multiparty protocol and we generate *correct-by-construction* APIs for developers to implement the protocol. The generated APIs use WebSocket to establish communication between participants, utilising its flexibility over the traditional HTTP model. When developers use our generated APIs to correctly implement the protocol endpoints, STScript guarantees the freedom from communication errors, including deadlocks, communication mismatches, channel usage violation or cancellation errors.

Our toolchain is backed by a new session theory, a *routed multiparty session types theory* (RouST), to endow servers with the capacity to *route messages* between web clients. The new theory addresses a practical limitation that WebSocket connections still require clients to connect to a prescribed server, constraining the ability for inter-client communication. To overcome this, our API *routes* inter-client messages through the server, improving the expressiveness over previous work and enabling developers to correctly implement multiparty protocols. In our travel agency scenario, the agency plays the server role: it will establish WebSocket channels with each participant, and be tasked with routing all the messages between the friends. We formalise this routing mechanism as RouST and prove deadlock-freedom of RouST and show a behaviour-preserving encoding from the original MPST to RouST. The formalism and results in RouST directly guide a deadlock-free protocol implementation in Node.js via the router, preserving communication structures of the original protocol written by a developer.

Finally, we evaluate our toolchain (STScript) by case studies. We evaluate the expressiveness by implementing a number of web applications, such as interactive multiplayer games (*Noughts and Crosses*, *Battleship*) and web services (*Travel Agency*) that require routed communication.

Contributions and Structure of the Paper.

§ 2 We present an overview of our toolchain STScript, for generating communication-safe web applications in TypeScript from multiparty protocol descriptions.
§ 3 We motivate how the generated code executes the multiparty protocol descriptions, and present how STScript prevents common errors in the context of web applications.
§ 4 We present RouST, multiparty session types (MPST) extended with *routing*, and define a trace-preserving encoding of the original MPST into RouST.
§ 5 We show the expressiveness of STScript via a case study.
§ 6 We give related and future work.

Supplementary material lists omitted code, definitions, performance benchmarks and detailed proofs, **appendix** in the paper refers to the supplementary material. STScript is available on GitHub (https://github.com/STScript-2020/STScript, anonymised). We shall submit code for the benchmark, case studies and STScript as our **artifact**.

2 Overview

In this section, we give an overview of our code generation toolchain STScript (Fig. 3), demonstrate how to implement the travel agency scenario (Fig. 1) as a TypeScript web application, and explain how STScript prevents those errors.

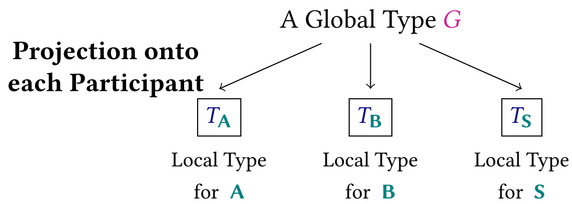

Figure 2. Top-down MPST Design Methodology

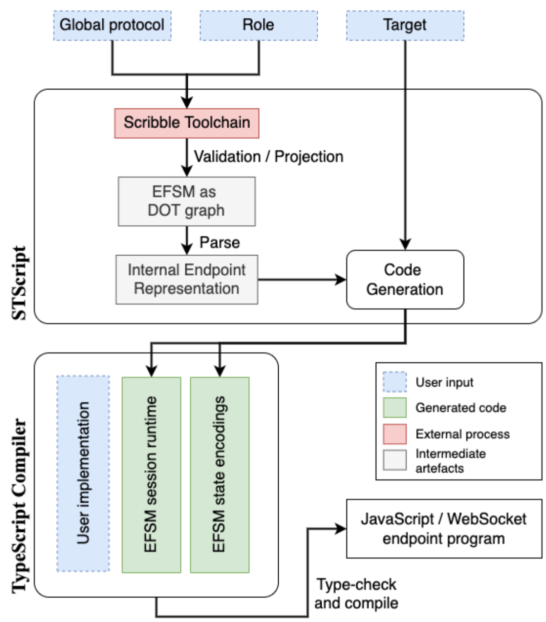

Figure 3. Overview of the toolchain STScript

Multiparty Session Type Design Workflow. Multiparty session types (MPST) [15] use a top-down design methodology (Fig. 2). Developers begin with *specifying* the global communication pattern of all participants in a *global type* or a *global protocol*. The protocol is described in the Scribble protocol description language [16, 29, 32]. We show the global protocol of the travel agency scenario (in § 1) in Fig. 4.

The Scribble language provides a user-friendly way to describe the global protocol in terms of a sequence of message exchanges between roles. A message is identified by its label (e.g. `Suggest`, `Query`, etc), and carries payloads (e.g. `number`, `string`, etc). The `choice` syntax (e.g. Line 4) describes possible branches of the protocol – in this case, the Server may respond to the query either with `Available`, so the customer continues booking, or with `Full`, so the customer retries by restarting the protocol via the `do` syntax (Line 13).

In this scenario, we designate the roles **A** and **B** as *client roles*, and role **S** as a *server role*. Participating endpoints can obtain their local views of the communication protocol, known as *local types*, via *projection* from the specified global type (Fig. 2). The local type of an endpoint can be then used in the code generation process, to generate APIs that are *correct by construction* [17, 20, 33].

```
1  global protocol TravelAgency(role A, role B, role S)
2  { Suggest(string) from B to A;  // friend suggests place
3    Query(string) from A to S;
4    choice at S
5       { Available(number) from S to A;
6         Quote(number) from A to B; // check quote with friend
7         choice at B
8            { OK(number) from B to A;
9              Confirm(credentials) from A to S; }
10        or  { No() from B to A;
11             Reject() from A to S; } }
12   or { Full() from S to A; Full() from A to B;
13        do TravelAgency(A, B, S); } }
```

Figure 4. Travel Agency Protocol in Scribble

The code generation toolchain STScript (Fig. 3) follows the MPST design philosophy. In STScript, we take the global protocol as inputs, and generate endpoint code for a given role as outputs, depending on the nature of the role. We use the Scribble toolchain for initial processing, and use an *endpoint finite state machine* (EFSM) based code generation technique targeting the TypeScript Language.

Targeting Web Programming. The TypeScript [2] programming language is used for web programming, with a static type system and a compiler to JavaScript. TypeScript programs follow a similar syntax to JavaScript, but may contain type annotations that are checked statically by the TypeScript type-checker. After type-checking, the compiler converts TypeScript programs into JavaScript programs, so they can be run in browsers and other hosts (e.g. Node.js).

To implement a wide variety of communication patterns, we use the *WebSocket* protocol [12], enabling bi-directional communication between the client and the server after connection. This contrasts with the traditional request-response model of HTTP, where the client needs to send a request and the server may only send a response after receiving the request. WebSockets require an endpoint to listen for connections and the other endpoint connecting. Moreover, clients, using the web application in a browser, may *only* start a connection to a WebSocket, and servers may *only* listen for new connections. The design of WebSocket limits the ability for two clients to communicate directly via a WebSocket (e.g. Line 2 in Fig. 4). STScript uses the server to *route* messages between client roles, enabling communication between all participants via a star network topology.

An important aspect of web programming is the interactivity of the user interface (UI). Viewed in a browser, the web application interacts with the user via UI events, e.g. mouse clicks on buttons. The handling of UI events may be implemented to send messages to the client (e.g. when the "Submit" button on the form is clicked), which may lead to practical problems. For instance, would clicking "Submit" button twice create two bookings for the customer? We use the popular *React.js* UI framework for generating client endpoints, and generate APIs that prevent such errors from happening.

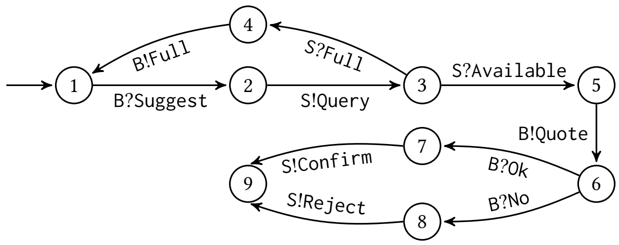

Figure 5. EFSM for TravelAgency role **A**

Callback-Style API for Clients and Servers. Our code generation toolchain STScript produces TypeScript APIs in a *callback-style* fashion [33] to *statically* guarantee channel linearity. The input global protocol is analysed by the toolchain for well-formedness, and an *endpoint finite state machine* (EFSM) is produced for each endpoint. We illustrate the EFSM for role **A** in Fig. 5. The states in the EFSM represent local types (subject to reductions) and transitions represent communication actions (The symbol ! stands for sending actions, ? stands for receiving actions).

In the callback API style, type signatures of callbacks are generated for transitions in the EFSM. Developers implement the callbacks to complete the program logic part of the application, whilst a generated *runtime* takes care of the communication aspects. For callbacks, sending actions correspond to callbacks prompting the payload type as a *return type*, so that the returned value can be sent by the runtime. Dually, receiving actions correspond to callbacks taking the payload type as an *argument*, so that the runtime invokes the callback with the received value.

Implementing the Server Role. In the travel agency protocol, as shown in Fig. 4, we designate role **S** as the server role. The server role does not only interact with the two clients, but also *routes* messages for the two clients. The routing will be handled automatically by the runtime, saving the need for developers to specify manually. As a result, the developer only handles the program logic regarding the server, in this use case, namely providing quotes for holiday bookings and handling booking confirmations.

```
1  import { Session, S } from "./TravelAgency/S";
2  const agencyProvider = (sessionID: string) => {
3    const handleQuery = Session.Initial({
4      Query: async (Next, dest) => {
5        // Provide quotes for holiday bookings
6        const res = await checkAvailability(sessionID, dest);
7        if (res.status === "available") {
8          return Next.Available([res.quote], Next => ...);
9        } else { return Next.Full([], handleQuery); } }, });
10   return handleQuery; };
```

All callbacks carry an extra parameter, `Next`, which acts as a *factory function* for constructing the successor state. This empowers IDEs to provide auto-completion for developers. For example, the factory function provided by the callback for handling a `Query` message (Line 4) prompts the permitted labels in the successor send state, as illustrated in Fig. 6.

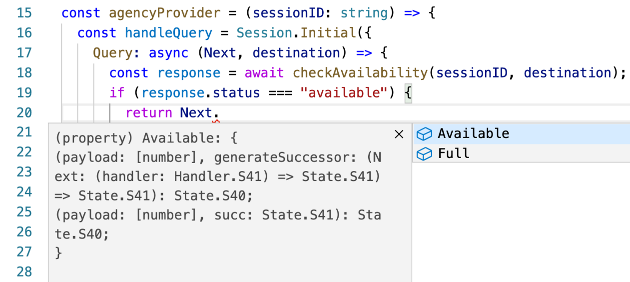

Figure 6. IDE Auto-Completion for Successor State

Implementing the Client Roles. To implement client roles, merely implementing the callbacks for the program logic is not sufficient — unlike servers, web applications have interactive user interfaces, additional to program logic. As mentioned previously, our code generation toolchain targets React.js for client roles. For background, the smallest building blocks in React.js are *components*, which can carry *properties* (immutable upon construction) and *states* (mutable). Components are *rendered* into HTML elements, and they are re-rendered when the component state mutates.

To bind the program logic with an interactive user interface, we provide *component factories* that allow the UI component to be interposed with the current state of the EFSM. Developers can provide the UI event handler to the component factory, and obtain a component for rendering. The generate code structure enforces that the state transition strictly follows the EFSM, so programmer errors (such as the double "submit" problem) are prevented by design.

```
1  render() {
2    const OK = this.OK('onClick', () => [this.state.split]);
3    const NO = this.No('onClick', () => []);
4    return (...
5      <NO><Button color='secondary'>No</Button></NO>
6      <OK><Button color='primary'>OK</Button></OK> ...); }
```

Using the send state component in the FSM for the endpoint **B** as an example, Line 2 reads, "generate a React component that sends the `OK` message with `this.state.split` as payload on a click event". It is used on Line 6 as a wrapper for a stylised `<Button>` component. The runtime invokes the handler and performs the state transition, which prevents the double "submit" problem by design.

Guaranteeing Communication Safety. Returning to the implementation in § 1, we outline how STScript prevents common errors to enable type-safe web programming.

Communication Mismatch All generated callbacks are typed according to the permitted payload data type specified in the protocol, making it impossible for traveller **A** to send the quote as a string by accident.

Channel Usage Violation The generated client-side runtime requires the developer to provide different UI components for each EFSM state – once traveller **A** submits a quote, the runtime will transition to, thus render the component of, a different EFSM state. This guarantees that, whilst waiting for a response from traveller **B**, it is impossible for traveller **A** to submit another quote and violate channel linearity.

Handling Session Cancellation If either traveller closes their browser before the protocol runs to completion, the generated runtimes leverage the events available on their WebSocket connections to notify (via the server) other roles about the session cancellation. The travel agency can implement the error handler callback (generated by STScript) to perform clean-up logic in response to cancellations.

3 Implementation

In this section, we explain how the generated code executes the EFSM for Node.js and React.js targets. We also present how STScript APIs handle errors in a dynamic web-based environment (for full code, see Appendix D).

Session Runtime. The purpose of the session runtime is to execute the EFSM in a manner permitted by the multiparty protocol description. The runtime keeps track of the current state, performs the required communication action (i.e. send or receive a message), and transitions to the successor state. The runtime provides *seams* for the developer to inject the callback implementations, which define application-specific concerns for the EFSM, such as what message payload to send (and dually, how to process a received message). This design conceals the WebSocket APIs from the developer and entails that the developer cannot trigger a send or receive action, so STScript can *statically* guarantee protocol conformance.

Executing the EFSM in Node.js. Each state of the EFSM is characterised by a (generated) `State` class and a type describing the shape of the callback (supplied by the developer). To allow the server to correctly manage concurrent sessions, the developer can access a (generated) *session ID* when implementing the callbacks. STScript also generates *IO interfaces* for each kind of EFSM state – send, receive, or terminal. The generated `State` class implements the interface corresponding to the type of communication action it performs.

```
next(state: State.Type) {
  switch (state.type) {
  case 'Send': return state.performSend(
    this.next, this.cancel, this.send);
  case 'Receive': return state.prepareReceive(
    this.next, this.cancel, this.registerMessageHandler);
  case 'Terminal': return; }}
```

The session runtime for Node.js is a class that executes the EFSM using a *state transition function* parameterised by the `State` class of the current EFSM state. As the IO interfaces constitute a *discriminated union*, the runtime can parse the type of the current EFSM state and propagate the appropriate IO functions (for sending or receiving) to the `State` class. In turn, the `State` class invokes the callback supplied by the developer to inject program logic into the EFSM, perform the communication action (using `this.send` or `this.registerMessageHandler`), and invoke the state transition function with the successor state.

Notably, the routed messages are completely absent because the generated code transparently routes messages without exposing any details. As messages specify their intended recipient, the runtime identifies messages not intended for the server by inspecting the metadata, and forwards them to the WebSocket connected to the intended recipient.

Executing the EFSM in React.js. Each state in the EFSM is encoded as an *abstract* React component. The developer implements the EFSM by extending the abstract classes to provide their own implementation – namely, to build their user interface. Components for send states can access *component factories* to generate React components that perform a send action when a UI event (e.g. `onClick`, `onMouseOver`) is triggered. Components for receive states must implement abstract methods to handle all possible incoming messages.

The session runtime for React.js is a React component, instantiated using the developer's implementation of each EFSM state. Channel communications are managed by the runtime, so the developer's implementations cannot access the WebSocket APIs, which prevents channel reuse by construction. The runtime renders the component of the current EFSM state and binds the permitted communication action through supplying component properties.

Error Handling. An error handling mechanism is critical for web applications. Clients can disconnect from the session due to network connectivity issues or simply by closing the browser. Similarly, servers may also face connectivity issues.

Upon instantiating the session runtime, STScript requires developers to supply a *cancellation handler* to handle *local exceptions* (e.g. errors thrown by application logic) and *global session cancellations* (e.g. disconnection events by another endpoint). The session runtime detects cancellation by listening to the *close event* on the WebSocket connection, and invokes the cancellation handler with appropriate arguments on a premature close event. We parameterise the cancellation handlers with additional information (e.g. which role disconnected from the session, the reason for the disconnection) to let developers be more specific in their error handling logic.

Cancellation Handlers for Servers. Server endpoints define cancellation handlers through a function, parameterised by the *session ID*, the *role* which initiated the cancellation, and (optionally) the *reason* for the cancellation — if the server-side logic throws an exception, the handler can access the thrown error through the `reason` parameter.

```
const handleCancel = async (sessionID, role, reason) => {
  if (role === Role.Self) {
    console.error(`${sessionID}: internal server error`); }
  else { await tryRelease(sessionID); }};
// Instantiate session runtime
new S(wss, handleCancel, agencyProvider);
```

Using the Travel Agency scenario introduced in § 1, if the customer prematurely closes their browser before responding to a `Quote`, the server can detect this (Line 4) and release the reservation to preserve data integrity.

Cancellation Handlers for Clients. Browser-side endpoints also define cancellation handlers through a function parameterised in the same way as those in Node.js, but must return a React component to be rendered by the session runtime. In the context of the Travel Agency scenario, the customer can render a different UI depending on whether the server disconnected or their friend closed their web browser prematurely. Browser endpoints can also respond to cancellations emitted by other client-side roles: when a browser endpoint disconnects, the server detects this and propagates the cancellation to the other client-side roles.

4 RouST: Routed Session Types

This section defines the syntax and semantics of RouST and proves some important properties. We show the sound and complete trace correspondence between a global type and a collection of endpoint types projected from the global type (Theorem 4.6). Using this result, we prove deadlock freedom (Theorem 4.7). We then show that, in spite of the added routed communications, RouST does not over-serialise communications by proving *communication preservations* between the original MPST and RouST (Theorem 4.11). These three theorems ensure that STScript endpoint programs are communication-safe, always make progress, and correctly conforms to the user-specified protocol.

4.1 Syntax of Routed Multiparty Session Types

We define the syntax of *global types* G and *local types* (or *endpoint types*) T in Definition 4.1. Global types are also known as *protocols* and describe the communication behaviour between all participating roles (participants), while local types describe the behaviour of a single participating role. We shade additions to the original (or *canonical*) multiparty session type (MPST) [9, 11, 15, 28] in this colour .

Definition 4.1 (Global and Local Types)**.** The syntax of *global* and *local types* are defined below:

$$
\begin{array}{llll}
G ::= \mathsf{end} \mid \mathbf{t} \mid \mu\mathbf{t}.G & T ::= \mathsf{end} \mid \mathbf{t} \mid \mu\mathbf{t}.T & \mid \mathbf{p} \hookrightarrow \mathbf{q} : \{l_i : T_i\}_{i \in I} \\
\quad \mid \mathbf{p} \to \mathbf{q} : \{l_i : G_i\}_{i \in I} & \quad \mid \mathbf{p} \oplus \{l_i : T_i\}_{i \in I} & \mid \mathbf{p} \oplus \langle \mathbf{q} \rangle \, \{l_i : T_i\}_{i \in I} \\
\quad \mid \mathbf{p} \text{--}\mathbf{s}\text{→}\, \mathbf{q} : \{l_i : G_i\}_{i \in I} & \quad \mid \mathbf{p} \,\&\, \{l_i : T_i\}_{i \in I} & \mid \mathbf{p} \,\&\, \langle \mathbf{q} \rangle \, \{l_i : T_i\}_{i \in I}
\end{array}
$$

Global Types. $\mathbf{p} \to \mathbf{q} : \{l_i : G_i\}_{i \in I}$ describes a ***direct communication*** of a message l_i from a role $\mathbf{p}$ to $\mathbf{q}$. We require that $\mathbf{p} \neq \mathbf{q}$, that labels l_i are pairwise distinct, and that the index set I is not empty. The message in the communication can carry a label among a set of permitted labels l_i and some payload. After a message with label l_i is received by $\mathbf{q}$, the communication continues with G_i, according to the chosen label. For simplicity, we do not include payload types (integers, strings, booleans, etc) in the syntax. We write $\mathbf{p} \to \mathbf{q} : l : G$ for single branches. For recursion, we adopt an *equi-recursive* view [25, §21], and use $\mu\mathbf{t}.G$ and $\mathbf{t}$ for a ***recursive protocol*** and a ***type variable***. We require that recursive types are *contractive (guarded)*, i.e. the recursive type $\mu\mathbf{t}.G$ progresses after the substitution $G[\mu\mathbf{t}.G/\mathbf{t}]$, prohibiting types such as $\mu\mathbf{t}.\mathbf{t}$. We use end to mark the ***termination*** of the protocol, and often omit the final end.

To support routed communication, we allow messages to be sent through a *router role*. A ***routed communication*** $\mathbf{p} \text{--}\mathbf{s}\text{→}\, \mathbf{q} : \{l_i : G_i\}_{i \in I}$ describes a router role $\mathbf{s}$ coordinating the communication of a message from $\mathbf{p}$ to $\mathbf{q}$: $\mathbf{q}$ offers $\mathbf{p}$ a choice in the index set I, but $\mathbf{p}$ sends the selected choice l_i to the router $\mathbf{s}$ instead. The router *forwards* the selection from $\mathbf{p}$ to $\mathbf{q}$. After $\mathbf{q}$ receives $\mathbf{p}$'s selection, the communication continues with G_i. $\mathbf{s}$ ranges over the set of roles $\mathbf{p}, \mathbf{q}, \cdots$, but we use $\mathbf{s}$ by convention as the router is usually some server. The syntax for routed communication shares the same properties as direct communication, but we additionally require that $\mathbf{p} \neq \mathbf{q} \neq \mathbf{s}$. We use $\mathtt{pt}(G)$ to denote the set of participants in the global type G.

Example 4.2 (Travel Agency)**.** The travel agency protocol, as shown in Fig. 4, is described by the global type G_{travel} in the original MPST, and G^R_{travel} in RouST.

$$
G_{\text{travel}} = \mu\mathbf{t}.\mathbf{B} \to \mathbf{A} : \mathit{Suggest} \,.\, \mathbf{A} \to \mathbf{S} : \mathit{Query} \,. \\
\mathbf{S} \to \mathbf{A} : \left\{ \begin{array}{l}
\mathit{Available} : \\
\quad \mathbf{A} \to \mathbf{B} : \mathit{Quote} \,.\, \mathbf{B} \to \mathbf{A} : \\
\qquad \left\{ \begin{array}{l} \mathit{OK} : \mathbf{A} \to \mathbf{S} : \mathit{Confirm} \\ \mathit{No} : \mathbf{A} \to \mathbf{S} : \mathit{Reject} \end{array} \right\} \\
\mathit{Full} : \mathbf{A} \to \mathbf{B} : \mathit{Full} \,.\mathbf{t}
\end{array} \right\}
$$

$$
G^R_{\text{travel}} = \mu\mathbf{t}.\mathbf{B} \text{--}\mathbf{S}\text{→}\, \mathbf{A} : \mathit{Suggest} \,.\, \mathbf{A} \to \mathbf{S} : \mathit{Query} \,. \\
\mathbf{S} \to \mathbf{A} : \left\{ \begin{array}{l}
\mathit{Available} : \\
\quad \mathbf{A} \text{--}\mathbf{S}\text{→}\, \mathbf{B} : \mathit{Quote} \,.\, \mathbf{B} \text{--}\mathbf{S}\text{→}\, \mathbf{A} : \\
\qquad \left\{ \begin{array}{l} \mathit{OK} : \mathbf{A} \to \mathbf{S} : \mathit{Confirm} \\ \mathit{No} : \mathbf{A} \to \mathbf{S} : \mathit{Reject} \end{array} \right\} \\
\mathit{Full} : \mathbf{A} \text{--}\mathbf{S}\text{→}\, \mathbf{B} : \mathit{Full} \,.\mathbf{t}
\end{array} \right\}
$$

Local Types. We first describe the local types in the original MPST theory. $\mathbf{q} \,\&\, \{l_i : T_i\}_{i \in I}$ stands for ***branching*** and $\mathbf{q} \oplus \{l_i : T_i\}_{i \in I}$ stands for ***selection***. From the perspective of $\mathbf{p}$, branching (resp. selection) offers (resp. selects) a choice among an index set I to (resp. from) $\mathbf{q}$, and communication continues with the corresponding T_i. Local types $\mu\mathbf{t}.T$, $\mathbf{t}$ and end have the same meaning as their global type counterparts.

We add new syntax constructs to express routed communication from the perspective of each role involved. The local type $\mathbf{p} \,\&\, \langle \mathbf{s} \rangle \, \{l_i : T_i\}_{i \in I}$ is a ***routed branching***: $\mathbf{q}$ is offering a choice from an index set I to $\mathbf{p}$ (the intended sender), but expects to receive $\mathbf{p}$'s choice via the router role $\mathbf{s}$; if the message received is labelled l_i, $\mathbf{q}$ will continue with local type T_i. The local type $\mathbf{q} \oplus \langle \mathbf{s} \rangle \, \{l_i : T_i\}_{i \in I}$ is a ***routed selection***: $\mathbf{p}$ makes a selection from an index set I to $\mathbf{q}$ (the intended recipient), but sends the selection to the router role $\mathbf{s}$; if the message sent is labelled l_i, $\mathbf{p}$ will continue with local type T_i. The local type $\mathbf{p} \hookrightarrow \mathbf{q} : \{l_i : T_i\}_{i \in I}$ is a ***routing communication***. The router role $\mathbf{s}$ orchestrates the communication from $\mathbf{p}$ to

$\mathbf{q}$, and continues with local type T_i depending on the label of the forwarded message. We keep track of the router role to distinguish between routing communications from normal selection and branching interactions.

Endpoint Projection. The local type T of a participant $\mathbf{p}$ in a global type G is obtained by the *endpoint projection* of G onto $\mathbf{p}$, denoted by G as $G \upharpoonright \mathbf{p}$.

Definition 4.3 (Projection)**.** The projection of G onto $\mathbf{r}$, written $G \upharpoonright \mathbf{r}$ is defined as:

$$(\mathbf{p} \xrightarrow{\mathbf{s}} \mathbf{q} : \{l_i : G_i\}_{i\in I}) \upharpoonright \mathbf{r} = \begin{cases} \mathbf{q} \oplus \langle \mathbf{s} \rangle \{l_i : G_i \upharpoonright \mathbf{r}\}_{i\in I} & \text{if } \mathbf{r} = \mathbf{p} \\ \mathbf{p} \,\&\, \langle \mathbf{s} \rangle \{l_i : G_i \upharpoonright \mathbf{r}\}_{i\in I} & \text{if } \mathbf{r} = \mathbf{q} \\ \mathbf{p} \hookrightarrow \mathbf{q} : \{l_i : G_i \upharpoonright \mathbf{r}\}_{i\in I} & \text{if } \mathbf{r} = \mathbf{s} \\ \bigsqcap_{i\in I} G_i \upharpoonright \mathbf{r} & \text{otherwise} \end{cases}$$

$$(\mu \mathbf{t}.G) \upharpoonright \mathbf{r} = \begin{cases} \mu \mathbf{t}.(G \upharpoonright \mathbf{r}) & \text{if } G \upharpoonright \mathbf{r} \neq \mathbf{t}' \\ \mathtt{end} & \text{otherwise} \end{cases} \qquad \mathtt{end} \upharpoonright \mathbf{r} = \mathtt{end} \qquad \mathbf{t} \upharpoonright \mathbf{r} = \mathbf{t}$$

The projection $(\mathbf{p} \to \mathbf{q} : \{l_i : G_i)\}_{i\in I} \upharpoonright \mathbf{r}$ is defined similar to $(\mathbf{p} \xrightarrow{\mathbf{s}} \mathbf{q} : \{l_i : G_i\}_{i\in I}) \upharpoonright \mathbf{r}$ dropping $\mathbf{s}$ (in the resulting local type) and the third case.

The rule uses the *merging operator* ($\sqcap$) when projecting a routed communication onto a non-participant. The operator checks that the projections of all continuations must be "compatible" (see Definition A.2).

Example 4.4 (Merging Local Types)**.** Two branching types from the same role with disjoint labels can merged into a type carrying both labels, e.g. $\mathbf{A} \,\&\, \mathit{Hello}.\mathtt{end} \sqcap \mathbf{A} \,\&\, \mathit{Bye}.\mathtt{end} = \mathbf{A} \,\&\, \{\mathit{Hello} : \mathtt{end}; \mathit{Bye} : \mathtt{end}\}$. The same is not true for selections, $\mathbf{A} \oplus \mathit{Hello}.\mathtt{end} \sqcap \mathbf{A} \oplus \mathit{Bye}.\mathtt{end}$ is undefined.

$$G_1 = \mathbf{A} \to \mathbf{B} : \begin{Bmatrix} \mathit{Greet} : \mathbf{A} \to \mathbf{C} : \mathit{Hello}\ .\mathtt{end} \\ \mathit{Farewell} : \mathbf{A} \to \mathbf{C} : \mathit{Bye}\ .\mathtt{end} \end{Bmatrix}$$

$$G_2 = \mathbf{A} \to \mathbf{B} : \begin{Bmatrix} \mathit{Greet} : \mathbf{C} \to \mathbf{A} : \mathit{Hello}\ .\mathtt{end} \\ \mathit{Farewell} : \mathbf{C} \to \mathbf{A} : \mathit{Bye}\ .\mathtt{end} \end{Bmatrix}$$

Consequently, the global type G_1 can be projected to role $\mathbf{C}$, but not G_2. Moreover, G_1 is well-formed, and G_2 is not.

Well-formedness. In the original theory, a global type G is *well-formed* (or *realisable*), denoted $\mathtt{wellFormed}(G)$, if the projection is defined for all its participants.

$$\mathtt{wellFormed}(G) \stackrel{\text{def}}{=} \forall \mathbf{p} \in \mathtt{pt}(G).\ G \upharpoonright \mathbf{p} \text{ exists}$$

We assume that the global type G is contractive (guarded).

In RouST, we say that a global type is well-formed *with respect to the role* $\mathbf{s}$ *acting as the router*. We define the characteristics that $\mathbf{s}$ must display in G to prove that it is a router, and formalise this as an *inductive* relation, $G \circledast \mathbf{s}$ (Definition 4.5), which reads $\mathbf{s}$ *is a centroid in* G. The intuition is that $\mathbf{s}$ is at the centre of all communication interactions.

Definition 4.5 (Centroid)**.** The relation $G \circledast \mathbf{s}$ ($\mathbf{s}$ is the centroid of G) is defined by the two axioms $\mathtt{end} \circledast \mathbf{s}$ and $\mathbf{t} \circledast \mathbf{s}$ and by the following rules:

$$\frac{G \circledast \mathbf{s}}{\mu \mathbf{t}.G \circledast \mathbf{s}} \qquad \frac{\mathbf{s} \in \{\mathbf{p}, \mathbf{q}\} \quad \forall i \in I.\ G_i \circledast \mathbf{s}}{\mathbf{p} \to \mathbf{q} : \{l_i : G_i\}_{i\in I} \circledast \mathbf{s}} \qquad \frac{\mathbf{r} = \mathbf{s} \qquad \forall i \in I.\ G_i \circledast \mathbf{s}}{\mathbf{p} \xrightarrow{\mathbf{r}} \mathbf{q} : \{l_i : G_i\}_{i\in I} \circledast \mathbf{s}}$$

For direct communication, $\mathbf{s}$ must be a participant and a centroid of all continuations. For routed communication, $\mathbf{s}$ must be the router and be a centroid of all continuations. Now we define of well-formedness of a global type G in RouST with respect to the router $\mathbf{s}$ (denoted $\mathtt{wellFormed}^R(G, \mathbf{s})$):

$$\mathtt{wellFormed}^R(G, \mathbf{s}) \stackrel{\text{def}}{=} (\forall \mathbf{p} \in \mathtt{pt}(G).\ G \upharpoonright \mathbf{p} \text{ exists}) \wedge G \circledast \mathbf{s}$$

4.2 Semantics of RouST

This subsection defines the labelled transition system (LTS) over global types for RouST, building upon [11].

First, we define the labels (actions) in the LTS which distinguish the *direct* sending (and reception) of a message from the sending (and reception) of a message *via* an intermediate routing endpoint. *Labels* range over $l, l', \cdots$ are defined by:

$$l ::= \mathbf{pq}!j \mid \mathbf{pq}?j \mid \mathtt{via}\langle \mathbf{s} \rangle(\mathbf{pq}!j) \mid \mathtt{via}\langle \mathbf{s} \rangle(\mathbf{pq}?j)$$

The label $\mathtt{via}\langle \mathbf{s} \rangle(\mathbf{pq}!j)$ represents the *sending* (performed by $\mathbf{p}$) of a message labelled j to $\mathbf{q}$ through the intermediate router $\mathbf{s}$. The label $\mathtt{via}\langle \mathbf{s} \rangle(\mathbf{pq}?j)$ represents the *reception* (initiated by $\mathbf{q}$) of a message labelled j send from $\mathbf{p}$ through the intermediate router $\mathbf{s}$. The *subject* of a label l, denoted by $\mathrm{subj}(l)$, is defined as: $\mathrm{subj}(\mathtt{via}\langle \mathbf{s} \rangle(\mathbf{pq}!j)) = \mathrm{subj}(\mathbf{pq}!j) = \mathbf{p}$; and $\mathrm{subj}(\mathtt{via}\langle \mathbf{s} \rangle(\mathbf{pq}?j)) = \mathrm{subj}(\mathbf{pq}?j) = \mathbf{q}$.

LTS Semantics over Global Types. The LTS semantics models *asynchronous communication* to reflect our implementation. We introduce intermediate states (i.e. messages in transit) within the grammar of global types: the construct $\mathbf{p} \rightsquigarrow \mathbf{q}.\ j : \{l_i : G_i\}_{i\in I}$ represents that the message l_j has been sent by $\mathbf{p}$ but not yet received by $\mathbf{q}$; and the construct $\mathbf{p} \underset{\mathbf{s}}{\rightsquigarrow} \mathbf{q}.\ j : \{l_i : G_i\}_{i\in I}$ represents that l_j has been sent from $\mathbf{p}$ to the router $\mathbf{s}$ *but not yet routed to* $\mathbf{q}$. We define the LTS semantics over global types, denoted by $G \xrightarrow{l} G'$, in Fig. 7. [Gr1] and [Gr2] model the emission and reception of a message; [Gr3] models recursions; [Gr4] and [Gr5] model causally unrelated transmissions — we only enforce the syntactic order of messages for the participants involved in the action l. [Gr6] and [Gr7] are analogous to [Gr1] and [Gr2] for describing routed communication, but uses the "routed in-transit" construct instead. [Gr8] and [Gr9] are analogous to [Gr4] and [Gr5]. An important observation from [Gr8] and [Gr9] is that, for the router, the syntactic order of routed communication can be freely interleaved between the syntactic order of direct communication. This is crucial to ensure that the router does not over-serialise communication. See Example 4.13 for an LTS example.

Relating Semantics of Global and Local Types. We prove the soundness and completeness of our LTS semantics with respect to projection. We take three steps following [11]:

1. We extend the LTS semantics with *configuration* $(\vec{T}, \vec{w})$, a collection of local types $\vec{T}$ with FIFO queues between each pair of participants $\vec{w}$.
2. We extend the definition of projection, to obtain a configuration of a global type (a *projected configuration*), which expresses intermediate communication over FIFO queues.

$$\frac{}{\mathbf{p} \to \mathbf{q} : \{l_i : G_i\}_{i\in I} \xrightarrow{\mathbf{pq}!j} \mathbf{p} \rightsquigarrow \mathbf{q}.\, j : \{l_i : G_i\}_{i\in I}}\ [\text{Gr1}]$$

$$\frac{}{\mathbf{p} \rightsquigarrow \mathbf{q}.\, j : \{l_i : G_i\}_{i\in I} \xrightarrow{\mathbf{pq}?j} G_j}\ [\text{Gr2}] \qquad \frac{G[\mu \mathbf{t}.G/\mathbf{t}] \xrightarrow{l} G'}{\mu \mathbf{t}.G \xrightarrow{l} G'}\ [\text{Gr3}]$$

$$\frac{\forall i \in I.\ G_i \xrightarrow{l} G'_i \qquad \mathrm{subj}(l) \notin \{\mathbf{p}, \mathbf{q}\}}{\mathbf{p} \to \mathbf{q} : \{l_i : G_i\}_{i\in I} \xrightarrow{l} \mathbf{p} \to \mathbf{q} : \{l_i : G'_i\}_{i\in I}}\ [\text{Gr4}]$$

$$\frac{G_j \xrightarrow{l} G'_j \qquad \mathrm{subj}(l) \neq \mathbf{q} \qquad \forall i \in I \setminus \{j\}.\ G'_i = G_i}{\mathbf{p} \rightsquigarrow \mathbf{q}.\, j : \{l_i : G_i\}_{i\in I} \xrightarrow{l} \mathbf{p} \rightsquigarrow \mathbf{q}.\, j : \{l_i : G'_i\}_{i\in I}}\ [\text{Gr5}]$$

$$\frac{}{\mathbf{p} \overset{\mathbf{s}}{\to} \mathbf{q} : \{l_i : G_i\}_{i\in I} \xrightarrow{\mathtt{via}\langle \mathbf{s}\rangle(\mathbf{pq}!j)} \mathbf{p} \underset{\mathbf{s}}{\rightsquigarrow} \mathbf{q}.\, j : \{l_i : G_i\}_{i\in I}}\ [\text{Gr6}]$$

$$\frac{}{\mathbf{p} \underset{\mathbf{s}}{\rightsquigarrow} \mathbf{q}.\, j : \{l_i : G_i\}_{i\in I} \xrightarrow{\mathtt{via}\langle \mathbf{s}\rangle(\mathbf{pq}?j)} G_j}\ [\text{Gr7}]$$

$$\frac{\forall i \in I.\ G_i \xrightarrow{l} G'_i \qquad \mathrm{subj}(l) \notin \{\mathbf{p}, \mathbf{q}\}}{\mathbf{p} \overset{\mathbf{s}}{\to} \mathbf{q} : \{l_i : G_i\}_{i\in I} \xrightarrow{l} \mathbf{p} \overset{\mathbf{s}}{\to} \mathbf{q} : \{l_i : G'_i\}_{i\in I}}\ [\text{Gr8}]$$

$$\frac{G_j \xrightarrow{l} G'_j \qquad \mathrm{subj}(l) \neq \mathbf{q} \qquad \forall i \in I \setminus \{j\}.\ G'_i = G_i}{\mathbf{p} \underset{\mathbf{s}}{\rightsquigarrow} \mathbf{q}.\, j : \{l_i : G_i\}_{i\in I} \xrightarrow{l} \mathbf{p} \underset{\mathbf{s}}{\rightsquigarrow} \mathbf{q}.\, j : \{l_i : G'_i\}_{i\in I}}\ [\text{Gr9}]$$

Figure 7. LTS over Global Types in RouST

3. We prove the trace equivalence between the global type and its projected configuration (i.e. the *initial configuration of* G, $(\vec{T}, \vec{\epsilon})$, where $\vec{T} = \{G \restriction \mathbf{p}\}_{\mathbf{p}\in\mathcal{P}}$ are a set of local types projected from G and ϵ is an empty queue).

The proof is non-trivial: due to space limitations, we omit the LTS semantics of local types, configurations and global configurations, and only state the main result (see Appendices A and B).

Theorem 4.6 (Sound and Complete Trace Equivalence)**.** *Let G be a well-formed canonical global type. Then G is trace equivalent to its initial configuration.*

Theorem 4.7 proves traces specified by a well-formed global protocol are *deadlock-free*, i.e. the global type either completes all communications, or otherwise makes progress. Note that this theorem implies the deadlock-freedom of configurations by Theorem 4.6.

Theorem 4.7 (Deadlock Freedom)**.** *Let G be a global type. Suppose G is well-formed with respect to some router $\mathbf{s}$, i.e.* $\mathtt{wellFormed}^R(G, \mathbf{s})$*. Then we have:*

$$\forall G'.\ \left(G \to^* G' \implies (G' = \mathsf{end}) \vee \exists G'', l.\ (G' \xrightarrow{l} G'')\right)$$

4.3 From Canonical MPST to RouST

We present an encoding from the canonical MPST theory (no routers) to RouST. This encoding is *parameterised* by the router role (conventionally denoted as $\mathbf{s}$); the intuition is that we encode all communication interactions to involve $\mathbf{s}$. If the encoding preserves the semantics of the canonical global type, then this encoding can guide a correct protocol implementation in Node.js via $\mathbf{s}$, preserving communication structures of the original protocol without deadlock.

Router-Parameterised Encoding. We define the router-parameterised encoding on global types, local types and LTS labels in the MPST theory. We start with global types, as presented in Definition 4.8. The main rule is the direct communication: if the communication did not go through $\mathbf{s}$, then the encoded communication involves $\mathbf{s}$ as the router.

Definition 4.8 (Encoding on Global Types)**.** The encoding of global type G with respect to the router role $\mathbf{s}$, denoted by $[\![G,\ \mathbf{s}]\!]$, is defined as:

$$[\![\mathsf{end},\ \mathbf{s}]\!] = \mathsf{end} \quad [\![\mathbf{t},\ \mathbf{s}]\!] = \mathbf{t} \quad [\![\mu\mathbf{t}.G,\ \mathbf{s}]\!] = \mu\mathbf{t}.[\![G,\ \mathbf{s}]\!]$$

$$[\![\mathbf{p} \to \mathbf{q} : \{l_i : G_i\}_{i\in I},\ \mathbf{s}]\!] = \begin{cases} \mathbf{p} \to \mathbf{q} : \{l_i : [\![G_i,\ \mathbf{s}]\!]\}_{i\in I} & \text{if } \mathbf{s} \in \{\mathbf{p}, \mathbf{q}\} \\ \mathbf{p} \overset{\mathbf{s}}{\to} \mathbf{q} : \{l_i : [\![G_i,\ \mathbf{s}]\!]\}_{i\in I} & \text{otherwise} \end{cases}$$

Local types express communication from the perspective of a particular role, hence the encoding takes two roles.

Definition 4.9 (Encoding on Local Types)**.** The encoding of local type T (from the perspective of role $\mathbf{q}$) with respect to the router role $\mathbf{s}$, denoted by $[\![T,\ \mathbf{q},\ \mathbf{s}]\!]$, is defined as:

$$[\![\mathsf{end},\ \mathbf{q},\ \mathbf{s}]\!] = \mathsf{end} \quad [\![\mathbf{t},\ \mathbf{q},\ \mathbf{s}]\!] = \mathbf{t} \quad [\![\mu\mathbf{t}.T,\ \mathbf{q},\ \mathbf{s}]\!] = \mu\mathbf{t}.[\![T,\ \mathbf{q},\ \mathbf{s}]\!]$$

$$[\![\mathbf{p} \oplus \{l_i : T_i\}_{i\in I},\ \mathbf{q},\ \mathbf{s}]\!] = \begin{cases} \mathbf{p} \oplus \{l_i : [\![T_i,\ \mathbf{q},\ \mathbf{s}]\!]\}_{i\in I} & \text{if } \mathbf{s} \in \{\mathbf{p}, \mathbf{q}\} \\ \mathbf{p} \oplus\langle\mathbf{s}\rangle\ \{l_i : [\![T_i,\ \mathbf{q},\ \mathbf{s}]\!]\}_{i\in I} & \text{otherwise} \end{cases}$$

$$[\![\mathbf{p}\ \&\ \{l_i : T_i\}_{i\in I},\ \mathbf{q},\ \mathbf{s}]\!] = \begin{cases} \mathbf{p}\ \&\ \{l_i : [\![T_i,\ \mathbf{q},\ \mathbf{s}]\!]\}_{i\in I} & \text{if } \mathbf{s} \in \{\mathbf{p}, \mathbf{q}\} \\ \mathbf{p}\ \&\langle\mathbf{s}\rangle\ \{l_i : [\![T_i,\ \mathbf{q},\ \mathbf{s}]\!]\}_{i\in I} & \text{otherwise} \end{cases}$$

Lemma 4.10 (Correspondence between Encodings)**.** *The projection of an encoded global type $[\![G,\ \mathbf{s}]\!] \restriction \mathbf{r}$ is equal to the encoded local type after projection $[\![G \restriction \mathbf{r},\ \mathbf{r},\ \mathbf{s}]\!]$, with respect to router $\mathbf{s}$, i.e. $\forall \mathbf{r}, \mathbf{s}, G.\ \left(\mathbf{r} \neq \mathbf{s} \implies [\![G,\ \mathbf{s}]\!] \restriction \mathbf{r} = [\![G \restriction \mathbf{r},\ \mathbf{r},\ \mathbf{s}]\!]\right)$.*

The constraint $\mathbf{r} \neq \mathbf{s}$ is necessary because we would otherwise lose information on the right-hand side of the equality: the projection of $\mathbf{s}$ in the original communication does not contain the routed interactions, so applying the local type encoding cannot recover this information.

Theorem 4.11 (Encoding Preserves Well-Formedness)**.** *Let G be a global type, and $\mathbf{s}$ be a role. Then we have:*

$$\mathtt{wellFormed}(G) \iff \mathtt{wellFormed}^R([\![G,\ \mathbf{s}]\!], \mathbf{s})$$

Preserving Communication. We present a crucial result that directly addresses the pitfalls of naive definitions of routed communication — our encoding does not over-serialise the original communication. We prove that our encoding preserves the LTS semantics over global types — or more precisely, we can use the encodings over global types and LTS actions to encode all possible transitions in the LTS for global types in the canonical MPST theory. We define the encoding of label l in the original MPST as: $[\![\mathbf{pq}!j, \mathbf{s}]\!] = \mathtt{via}\langle\mathbf{s}\rangle(\mathbf{pq}!j)$ and $[\![\mathbf{pq}?j, \mathbf{s}]\!] = \mathtt{via}\langle\mathbf{s}\rangle(\mathbf{pq}?j)$ if $\mathbf{s} \notin \{\mathbf{p}, \mathbf{q}\}$ and otherwise $[\![l, \mathbf{s}]\!] = l$.

Theorem 4.12 (Encoding Preserves Semantics). *Let G, G' be well-formed global types such that $G \xrightarrow{l} G'$ for some label l. Then we have:*

$$\forall l, \mathbf{s}.\ \left(G \xrightarrow{l} G' \iff [\![G, \mathbf{s}]\!] \xrightarrow{[\![l,\, \mathbf{s}]\!]} [\![G', \mathbf{s}]\!]\right)$$

We conclude with an example which demonstrates global semantics in RouST and a use of the encoding.

Example 4.13 (Encoding Preserves Semantics). Consider the global type

$$G = \mathbf{p} \to \mathbf{q} : M1 \,.\, \mathbf{s} \to \mathbf{q} : M2 \,.\, \mathtt{end}.$$

We apply our encoding with respect to the router role $\mathbf{s}$:

$$[\![G, \mathbf{s}]\!] = \mathbf{p} \xrightarrow{\mathbf{s}} \mathbf{q} : M1 \,.\, \mathbf{s} \to \mathbf{q} : M2 \,.\, \mathtt{end}.$$

We note that $l = \mathbf{sq}!M2$ can reduce G through [Gr1] (via one application of [Gr4]). After encoding, we have that $[\![l, \mathbf{s}]\!] = l$. The encoded global type $[\![G, \mathbf{s}]\!]$ can be reduced by l through [Gr1] (via one application of [Gr8]), as demonstrated by Theorem 4.12. The label $l = \mathbf{sq}!M2$ is a prefix of a valid execution trace for G, given below.

$$G \xrightarrow{\mathbf{sq}!M2} \xrightarrow{\mathbf{pq}!M1} \xrightarrow{\mathbf{pq}?M1} \xrightarrow{\mathbf{sq}?M2} \mathtt{end}$$

Interested readers can verify that the encoded trace (given below) is a valid execution trace for $[\![G, \mathbf{s}]\!]$.

$$[\![G, \mathbf{s}]\!] \xrightarrow{\mathbf{sq}!M2} \xrightarrow{\mathtt{via}\langle\mathbf{s}\rangle(\mathbf{pq}!M1)} \xrightarrow{\mathtt{via}\langle\mathbf{s}\rangle(\mathbf{pq}?M1)} \xrightarrow{\mathbf{sq}?M2} \mathtt{end}$$

5 Case Study: Noughts and Crosses Game

In this section, we demonstrate the expressiveness and applicability of STScript for modern web programming. We walk through how to implement *Noughts and Crosses* game with our toolchain, showing how the generated APIs prevent common errors. We choose this game since we can demonstrate the main features of STScript within the limited space. In Appendix C.2, we include performance benchmarks and additional larger cases studies: *Battleship*, another game with more complex program logic; and *Travel Agency* (Fig. 1), demonstrating the full feature of RouST.

Overview. We present the classic two-player turn-based game of *Noughts and Crosses*[2] here — see Appendix C.2 for other case studies. We formalise the game interactions using a Scribble protocol: both players, identified by *noughts (O's)* or *crosses (X's)* respectively, take turns to place a mark on an unoccupied cell of a 3-by-3 grid, until a player wins (when their markers form a straight line on the board) or a stalemate is reached (when all cells are occupied and no one wins).

[2]Deployed as http://stscript-noughts-and-crosses.herokuapp.com/

```
// Position on game board
type <typescript> "Coordinate" from "./Types" as Pt;
global protocol Game(role Svr, role P1, role P2) {
  Pos(Pt) from P1 to Svr;
  choice at Svr
    { Lose(Pt) from Svr to P2; Win(Pt) from Svr to P1; }
  or { Draw(Pt) from Svr to P2; Draw(Pt) from Svr to P1; }
  or { Update(Pt) from Svr to P2; Update(Pt) from Svr to P1;
       do Game(Svr, P2, P1); }} // Players swap turns
```

Game Server. We set up the game server as an Express.js application on top of the Node.js runtime. We define our own game logic in a Board class to keep track of the game state and expose methods to query the result. When the server receives a move, it notifies the game logic to update the game state *asynchronously* and return the game result caused by that move. The expressiveness of STScript enable the developer to define the handlers as async functions to use the game logic API correctly – this is prevalent in modern web programming, but not directly addressed in [13, 20].

The generated session runtime for Node.js is given as:

```
const gameManager = (gameID: string) => {
  const handleP1Move = Session.Initial({
    Pos: async (Next, move: Point) => {
      // Update current game with new move, return result
      switch (await DB.attack(gameID, 'P1', move)) {
      case MoveResult.Win:
        // Send losing result to P2, winning result to P1
        return Next.Lose([move], Next => (
          Next.Win([move], Session.Terminal)));
      case MoveResult.Draw: ...
      case MoveResult.Continue:
        // Notify both players and proceed to P2's turn
        return Next.Update([move], Next => (
          Next.Update([move], handleP2Move)) }});
  const handleP2Move = ...  // defined similarly
  return handleP1Move; }
// Initialise game server
new Svr(wss, handleCancellation, gameManager);
```

The runtime is initialised by a function parameterised by the *session ID* and returns the initial state. The developer can use the session ID as an identifier to keep track of concurrent sessions and update the board of the corresponding game.

Game Players. On the browser side, the main implementation detail for game players is to make moves. Intuitively, the developer implements a grid and binds a mouse click handler for each vacant cell to send its coordinates in a Pos(Point) message to the game server. Without STScript, developers need to synchronise the UI with the progression of protocol *manually* — for instance, they need to guarantee that the game board is *inactive* after the player makes a move, and manual efforts are error-prone and unscalable.

The generated React APIs from STScript make this intuitive, and guarantees communication safety in the meantime. By providing *React component factories* for send states, the

APIs let the developer trigger the same send action on multiple UI events with possibly different payloads. In the context of Noughts and Crosses, for each vacant cell on the game board, we create a `<SelectCell>` React component from the component factory function (Line 6). The factory builds a component that sends the `Pos` message with `x-y` coordinate as payload when the user clicks on it. We bind the `onClick` event to the table cell by wrapping it with the `<SelectCell>` component.

```
1 {board.map((row, x) => (<tr>
2 {row.map((cell, y) => {
3   const tableCell = <td>{cell}</td>;
4   if (cell === Cells.VACANT) {
5     const makeMove = (ev: React.MouseEvent) => ({ x, y });
6     const SelectCell = this.props.Pos('onClick', makeMove);
7     return <SelectCell>{tableCell}</SelectCell>; }
8   else { return tableCell; }})} </tr>)}
```

The session cancellation handler allows the developer to render useful messages to the player by making *application-specific* interpretations of the cancellation event. For example, if the opponent disconnects, the event can be interpreted as a forfeiture and a winning message can be rendered.

6 Related and Future Work

There are a vast number of studies on theories of session types [19], some of which are integrated in programming languages [1], or implemented as tools [30]. Here we focus on the most closely related work: **(1)** code generation from session types; **(2)** web applications based on session types; and **(3)** encoding multiparty sessions into binary connections.

Code Generation from Session Types. In general, a code generation toolchain takes a protocol (session type) description (in a domain specific language) and produces *well-typed APIs* conforming to the protocol. The Scribble [29, 32] language is widely used to describe multiparty protocols, agnostic to target languages. The Scribble toolchain implements the projection of global protocols, and the construction of endpoint finite state machines (EFSM). Many implementations use an EFSM-based approach to generate APIs for target programming languages, e.g. Java [17], Go [7], and F# [23], for distributed applications. Our work also falls into this category, where we generate *correct-by-construction* TypeScript APIs, but focusing on interactive web applications. Following [33], we generate callback-style APIs, adapted to fit the event-driven paradigm in web programming.

Alternatively, Demangeon et al. [10] propose MPST-based *runtime monitors* to *dynamically* verify protocol conformance, also available from code generation. Whilst a runtime approach is viable for JavaScript applications, our method, which leverages the TypeScript type system to *statically* provide communication safety to developers, gives a more rigorous guarantee. Ng et al. [24] propose a different kind of MPST-based code generation, where sequential C code can be parallelised according to a global protocol using MPI.

Session-Typed Web Development. Fowler [13] integrates *binary* session types into web application development. Our work encodes *multiparty* session types for web applications, *subsuming* binary sessions. King et al. [20] extend the Scribble toolchain for web applications targeting PureScript [26], a functional web programming language. In their work, a client may only communicate with one *designated* server role, whereas our work addresses this limitation via *routing* through a designated role. Jolie [22, 31] is a programming language designed for web services, capable of expressing multiparty sessions. Jolie extends the concept of choreography programming [5], where a choreography contains behaviour of all participants, and endpoints are derived directly from projections. Our work implements each endpoint separately. Moreover, we generate server and client endpoints using different styles to better fit their use case. Note that Links [8], PureScript [26] and Jolie [31] are not usually considered mainstream in modern web programming, whereas our tool targets popular web programming technologies.

Encoding of Multiparty Session Types. RouST models an "orchestrating" role (the router) for forwarding messages between roles, and this information is used to directly guide STScript to correctly implement the protocol in Node.js. The use of a *medium* process to encode multiparty into binary session types has been studied in theoretical settings, in particular, linear logic based session types [3, 4, 6]. In their setting, one medium process is used for orchestrating the multiparty communications between all roles in binary session types.Our encoding models the nature of web applications running over WebSockets, where browser clients can only directly connect to a server, not other clients.

Scalas et al. [27] show a different encoding of multiparty session types into linear π-calculus, which decomposes a multiparty session into binary channels *without* a medium process. This encoding is used to implement MPST with binary session types in Scala. Their approach uses *session delegation*, i.e. passing channels, which is difficult to implement with WebSockets. Our RouST focuses on modelling the routing mechanism at the *global types level*, so that our encoding can directly guide correct practical implementations.

Conclusion and Future Work. We explore the application of session types to modern interactive web programming, by using code generation to generate communication-safe APIs from a multiparty protocol specification. We incorporate routing semantics to seamlessly adapt MPST to address the practical challenges of using WebSocket protocols. Our approach integrates with popular industrial frameworks, and is backed by our theory of routed multiparty session types that guarantees communication safety.

For future work, we would like to extend **(1)** STScript with additional practical extensions of MPST, e.g. explicit connections [18], **(2)** our code generation approach to implement typestates in TypeScript, inspired by [21].

References

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2–3 (July 2016), 95–230. https://doi.org/10.1561/2500000031

[2] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

[3] Luís Caires and Jorge A. Pérez. 2016. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In *Formal Techniques for Distributed Objects, Components, and Systems*, Elvira Albert and Ivan Lanese (Eds.). Springer International Publishing, Cham, 74–95.

[4] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Sc huermann, and Philip Wadler. 2016. Coherence Generalises Duality: a logical explanation of multipart y session types. In *CONCUR'16 (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 59)*. Schloss Dagstuhl, 33:1–33:15.

[5] Marco Carbone and Fabrizio Montesi. 2013. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/2429069.2429101

[6] Marco Carbone, Fabrizio Montesi, Carsten Schormann, and Nobuko Yoshida. 2015. Multiparty Session Types as Coherence Proofs. In *CONCUR 2015 (LIPIcs, Vol. 42)*. Schloss Dagstuhl, 412–426.

[7] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 29 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290342

[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296.

[9] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *MSCS* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

[10] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design* 46, 3 (Jun 2015), 197–225. https://doi.org/10.1007/s10703-014-0218-8

[11] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming (LNCS, Vol. 7966)*. Springer, 174–186. a full version: http://arxiv.org/abs/1304.1902.

[12] Ian Fette and Alexey Melnikov. 2011. *The WebSocket Protocol*. RFC 6455. RFC Editor. 1–71 pages. https://www.rfc-editor.org/info/rfc6455

[13] Simon Fowler. 2020. Model-View-Update-Communicate: Session Types Meet the Elm Architecture. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.14

[14] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63 (2016), 1–67. Issue 1-9. https://doi.org/10.1145/2827695

[16] Raymond Hu. 2017. Distributed Programming Using Java APIs Generated from Session Types. *Behavioural Types: from Theory to Tools* (2017), 287–308.

[17] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering (LNCS, Vol. 9633)*. Springer, Berlin, Heidelberg, 401–418. https://doi.org/10.1007/978-3-662-49665-7_24

[18] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *Fundamental Approaches to Software Engineering*, Marieke Huisman and Julia Rubin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 116–133.

[19] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (2016). https://doi.org/10.1145/2873052

[20] Jonathan King, Nicholas Ng, and Nobuko Yoshida. 2019. Multiparty Session Type-safe Web Development with Static Linearity. In Proceedings *Programming Language Approaches to Concurrency- and Communication-cEntric Software,* Prague, Czech Republic, 7th April 2019 *(Electronic Proceedings in Theoretical Computer Science, Vol. 291)*, Francisco Martins and Dominic Orchard (Eds.). Open Publishing Association, 35–46. https://doi.org/10.4204/EPTCS.291.4

[21] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking Protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming* (Edinburgh, United Kingdom) *(PPDP '16)*. Association for Computing Machinery, New York, NY, USA, 146–159. https://doi.org/10.1145/2967973.2968595

[22] Fabrizio Montesi. 2016. Process-aware web programming with Jolie. *Science of Computer Programming* 130 (2016), 69 – 96. https://doi.org/10.1016/j.scico.2016.05.002

[23] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. 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* (Vienna, Austria) *(CC 2018)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/3178372.3179495

[24] Nicholas Ng, Jose G.F. Coutinho, and Nobuko Yoshida. 2015. Protocols by Default: Safe MPI Code Generation based on Session Types. In *CC 2015 (LNCS, Vol. 9031)*. Springer, 212–232.

[25] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[26] PureScript. 2020. *purescript/purescript*. PureScript. Accessed on 10th August 2020.

[27] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[28] Alceste Scalas and Nobuko Yoshida. 2019. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 30 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290343

[29] Scribble Authors. 2015. Scribble: Describing Multi Party Protocols. http://www.scribble.org/.

[30] António Ravara Simon Gay (Ed.). 2017. *Behavioural Types: from Theory to Tools.* River Publisher. https://www.riverpublishers.com/research_details.php?book_id=439

[31] The Jolie Team. 2020. Jolie Programming Language – Official Website. https://jolie-lang.org/. Accessed on 11th November 2020.

[32] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2014. The Scribble Protocol Language. In *8th International Symposium on Trustworthy Global Computing - Volume 8358* (Buenos Aires, Argentina) *(TGC 2013)*. Springer-Verlag, Berlin, Heidelberg, 22–41. https://doi.org/10.1007/978-3-319-05119-2_3

[33] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically Verified Refinements for Multiparty Protocols. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 148 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428216