Implementing Multiparty Session Types in Rust

Nicolas Lagaillardie[1(✉)], Rumyana Neykova[2(✉)], and Nobuko Yoshida[1(✉)]

[1] Imperial College London, London, UK
{n.lagaillardie19,n.yoshida}@imperial.ac.uk
[2] Brunel University London, London, UK
rumyana.neykova@brunel.ac.uk

Abstract. Multiparty Session Types (MPST) is a typing discipline for distributed protocols, which ensures communication safety and deadlock-freedom for more than two participants. This paper reports on our research project, implementing multiparty session types in Rust. Current Rust implementations of session types are limited to binary (two-party communications). We extend an existing library for binary session types to MPST. We have implemented a simplified Amazon Prime Video Streaming protocol using our library for both shared and distributed communication transports.

1 Introduction

In the last decade, the software industry has seen a shift towards programming languages that promote the coordination of concurrent and/or distributed software components through the exchange of messages over *communication channels*. Languages with native message-passing primitives (e.g., Go, Elixir and Rust) are becoming increasingly popular. In particular, Rust has been named the most loved programming language in the annual Stack Overflow survey for four consecutive years (2016–19)[1].

The advantage of message-passing concurrency is well-understood: it allows cheap horizontal scalability at a time when technology providers have to adapt and scale their tools and applications to various devices and platforms. Message-passing based software, however, is as vulnerable to errors as other concurrent programming techniques [16]. Much academic research has been done to develop rigorous theoretical frameworks for verification of message-passing programs. One such framework is *multiparty session types* (MPST) [5] – a type-based discipline that ensures that concurrent and distributed systems are *safe by design*. It guarantees that message-passing processes following a predefined communication protocol, are free from communication errors and deadlocks.

[1] https://insights.stackoverflow.com/survey/2019.

S. Bliudze and L. Bocchi (Eds.): COORDINATION 2020, LNCS 12134, pp. 127–136, 2020.
https://doi.org/10.1007/978-3-030-50029-0_8

Rust is a particularly appealing language for the practical embedding of session types. Its *affine type system* allows for static typing of linear resources – an essential requirement for the safety of session type systems. Rust combines efficiency with message-passing abstractions, thread and memory safety [15], and has been used for the implementation of large-scale concurrent applications such as the Mozilla browser, Firefox, and the Facebook blockchain platform, Libra. Despite the interest in the Rust community for verification techniques handling multiple communicating processes[2], the existing Rust implementations [8,9] are limited to *binary* (two-party) session types.

In this short paper, we present our design and implementation for multiparty session types in Rust. Our design follows a state-of-the-art encoding of multiparty into binary session types [13]. We generate local types in Rust, utilising the Scribble toolchain [12,18]. Our library for MPST programming in Rust, `mpst-rust`, is implemented as a thin wrapper over an existing binary session types library [9]. Differently from other MPST implementations that check the linear usage of channels at runtime (e.g. [6,13]), we rely on the Rust affine type system to type-check MPST programs. In addition, since we generate the local types from a readable global specification, errors caused by an affine (and not linear) usage of channels, a well-known limitation of the previous libraries [8,9], are easily avoided.

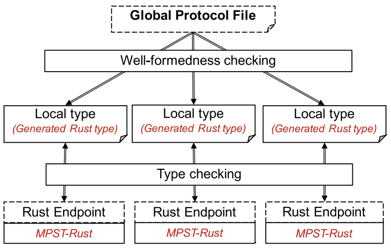
This paper is organised as follows: Sect. 2 gives an overview of our framework with a usecase; Sect. 3 shows our implementation and discusses the advantages of our approach; and Sect. 4 concludes with related and future work. Our library is available from https://github.com/NicolasLagaillardie/mpst_rust_github.

2 From Binary to Multiparty Sessions in Rust

Framework Overview: MPST in Rust. Our design resembles the top-down methodology of multiparty session types, as illustrated in Fig. 1. It follows three main steps [5,17]. First, a *global type*, also called a *global protocol*, is defined as a shared contract between communicating endpoint processes. A global protocol is then *projected* to each endpoint process, resulting in a *local type*. A local type involves only the interactions specific to a given endpoint. Finally, each endpoint process is type-checked against its projected local type.

The specific parts of our framework that distinguish it from other state-of-the-art MPST works are highlighted in *red*, which corresponds to our new library for MPST programming in Rust, `mpst-rust`. It is realised as a thin wrapper on top of an existing Rust library for validation of binary (2-party-only) session types. Developers use the MPST primitives provided by `mpst-rust` to implement endpoint programs. Also, our framework allows the types for each communication primitive to be either (1) generated from the Scribble toolchain; or (2) written by the developers. The Scribble toolchain [18] provides facilities for writing, verifying and projecting global protocols. Our framework guarantees

[2] https://users.rust-lang.org/t/anybody-working-on-multiparty-session-types-for-rust/10610.



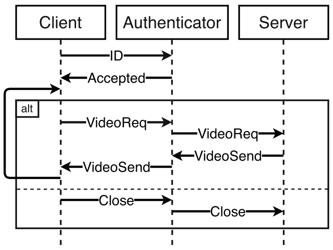


Fig. 1. MPST Workflow (left) and Amazon Prime Video usecase (right)

that processes implemented using `mpst-rust` primitives with Scribble-generated types are free from deadlocks, reception errors, and protocol deviations. Next, we explain, via an example, how the framework of MPST can be applied to Rust.

Example: Amazon Prime Video Streaming. The Amazon Prime Video streaming service is a usecase which can take full advantage of multiparty session types. Each streaming application connects to servers, and possibly other devices, to access services, and follows some specific protocol. To present our design, we use a simplified version of the protocol, illustrated in the diagram in Fig. 1 (right). The diagram should be read from top to bottom. The protocol involves three services – an `Authenticator` service, a `Server` and a `Client`. At first, `Client` connects to `Authenticator` by providing an identifying `id`. If the `id` is accepted, the session continues with a choice on `Client` to either request a video or end the session. The first branch is, *a priori*, the main service provided by Amazon Prime Video. `Client` cannot directly request videos from `Server`, and has to go through `Authenticator` instead. On the diagram, the choice is denoted as the frame `alt` and the choices are separated with the horizontal dotted line. The protocol is recursive, and `Client` can request new videos as many times as needed. The arrow going back on `Client` side in Fig. 1 represents this recursive behaviour. To end the session, `Client` first sends `Close` message to `Authenticator`, which then subsequently sends a `Close` message to `Server`.

Implementing the Authenticator role Using `mpst-rust`. Due to space limitations, we only show the implementation of the `Authenticator` role (hereafter role `A`), the implementations of the other roles (role `B` for the `Server` and role `C` for the `Client`) are similar. The Rust code for role `A` using the `mpst-rust` library is given in Fig. 2 (left). It closely follows the local protocol in Fig. 2 (right), that is projected from the global protocol by the Scribble toolchain. First, line 1 declares a function `authenticator` that is parametric in a multiparty channel `s` of type `VideoP_A`. The type `VideoP_A` specifies which operations are allowed on `s`. This type can either be written by the developer, or generated by Scribble (cf. Listing 1).

On line 3, `A` receives an identifying `id` from `C`. The function `recv_mpst_a_to_c`, provided by `mpst-rust` library returns the received value (the `id`) and the new multiparty channel, to be used in subsequent communications. Line 3 rebinds

```
 1  fn authenticator(s: VideoP_A<i32>)->              local protocol VideoP at A(
 2  Result<(), Box<Error>> {                          role B, role C)
 3   let (id, s) = recv_mpst_a_to_c(s)?;              { Declare(int) from C;
 4   let s = send_mpst_a_to_c(id + 1, s);              Accept(string) to C;
 5   let result = authenticator_recurs(s)?;            do VideoPRec(B, C);
 6   Ok(result)
 7  }                                                 }
 8
 9  fn authenticator_recurs(                          local protocol VideoPRec
10    s: VideoPRec_A<i32>)                             at A(role B, role C){
11  -> Result<(), Box<Error>> {                        choice at C {
12  offer_mpst_a_to_c!(s,
13  ChoiceA::Video(
14    SessionMpst{ch_ab, ch_ac, q})
15  => {                                                VideoRequest(
16    let s = SessionMpst{ch_ab,ch_ac, q}                string) from C;
17    let (req, s) = recv_mpst_a_to_c(s)?;              VideoRequest(
18    let s = send_mpst_a_to_b(req, s);                  string) to B;
19    let (v, s) = recv_mpst_a_to_b(s)?;                SendVideo(Video) from B;
20    let s = send_mpst_a_to_c(v, s);                   SendVideo(Video) to C;
21    authenticator_recurs(s)},                         do VideoPRec(B, C);
22  ChoiceA::End(SessionMpst{ch_ab, ch_ac, q})         } or {
23  => {
24    let s = SessionMpst{ch_ab, ch_ac, q};             Close() from C;
25    close_mpst(s)?;                                   Close() to B, C;
26    Ok(())})?;...}                                  }}
```

Fig. 2. Rust implementation of role A (left) and its local Scribble protocol (right)

the multiparty channel `s` with the new channel that is returned. Then, on line 4, we send back the answer to `C`, by utilising another `mpst-rust` communication primitive, `send_mpst_a_to_c`. The variable `s` is rebound again to the newly returned multiparty channel. Note that although the name of the function, `send_mpst_a_to_c`, suggests a binary communication, the function operates on a multiparty channel `s`. Our implementation follows the encoding, presented in [13], which encodes a multiparty channel as an indexed tuple of binary channels. Internally, `send_mpst_a_to_c` extracts from `s` the binary channel established between `A` and `C` and uses it for sending.

Lines 9–26 proceeds by implementing the recursive part of the protocol. The implementation of `authenticator_recurs` realises an internal choice – `A` can either receive a `VideoRequest` or a `Close`. This behaviour is realised by the `mpst-rust` macro `offer_mpst_a_to_c!` (line 12), which is applied to a multiparty channel `s` of a sum type between `ChoiceA::Video` and `ChoiceA::End`. The behaviour of each branch in the protocol is implemented as an anonymous function. For example, code in lines 13–21 supplies an anonymous function that implements the behaviour when `C` sends a `VideoRequest`, while lines 22–26 handle the `Close` request. Finally, `close_mpst(s)` closes all binary channels stored inside `s`. The types of the multiparty channel, as well as the generic types in the declaration of the `mpst-rust` communication functions, enable compile-time detection of protocol violations, such as swapping line 3 and line 4, using another communication primitive or using the wrong payload type.

```rust
1  /// Binary session types for A and C
2  type InitA<N> = Recv<N, Send<N,
3   RecvChoice<N>>>;
4  type RecvChoice<N> =
5    Recv<ChoiceA<N>, End>;
6  type AtoCVideo<N> =
7    Recv<N, Send<N, RecvChoice<N>>
8  type AtoCClose = End;
9
10
11 /// Binary session types for A and B
12 type AtoBVideo<N> =
13  Send<N, Recv<N, End>>;
14 type AtoBClose = End;
15
16 /// Declare usage order of channels
17 type QueueAInit = RoleAtoC<RoleAtoC<
18  RoleAtoC<RoleEnd>>>;
19 type QueueAChoice = RoleAtoC<RoleEnd>;
20 /// Declare usage order of channels
21 type QueueAVideo =
22  RoleAtoC<RoleAtoB<RoleAtoB<
23  RoleAtoC<RoleAtoC<RoleEnd>>>>>;
24 type QueueAClose = RoleEnd;
25
26 /// Declare MPST
27 type VideoP_A<N> = SessionMpst<End,
28  InitA<N>, QueueAInit>;
29
30 type VideoPRec_A<N> = SessionMpst<
31  End, RecvChoice<N>, QueueAChoice>;
32
33 enum ChoiceA<N> {
34  Video(SessionMpst<AtoBVideo<N>,
35   AtoCVideo<N>, QueueAVideo>),
36  End(SessionMpst<AtoBClose,
37   AtoCClose, QueueAClose>)
38 }
```

Listing 1. Local Rust types for role A

Typing the Authenticator Role. The types for the `Authenticator` role, used in Fig. 2 (left), are given in Listing 1. These types can be either written by the developer or generated from a global protocol, written in Scribble. Reception error safety is ensured since the underlying `mpst-rust` library checks *statically* that all pairs of binary types are dual to each other. Deadlock-freedom is ensured only if types are generated from Scribble since this guarantees that types are projected from a well-formed global protocol.

Next, we explain a type declaration for the `Authenticator` role. Lines 27–37 specify the three `SessionMpst` types which correspond to the types of the session channels used in Fig. 2 (left) – types `VideoP_A` (line 1), `Video_PRec_A` (line 9), and the types used inside the `offer` construct – `ChoiceA::Video` (line 13), and `ChoiceA::End` (line 22).

In the encoding of [13], which underpins `mpst-rust`, a multiparty channel is represented as an indexed tuple of binary channels. This is reflected in the implementation of `SessionMpst`, which is parameterised on the required binary session types. For example, the `VideoP_A<N>` takes as a parameter the binary types between `A` and `C`, and between `A` and `B`. At the beginning of the protocol (lines 1–7 in Fig. 2 (left)) `B` and `A` do not interact, hence the binary type for `B` is `End`. The type `InitA<N>` (line 2 in Listing 1) specifies the behaviour between `A` and `C`, notably that `A` first receives a message, then it sends a message, and later it continues as the type `RecvChoice<N>`. The binary session types between `A` and `B`, and between `A` and `C` are given in lines 12–14 and lines 2–9 respectively; we use the primitives declared in the existing binary session types library [9]. The generic parameter `N` refers to a `trait` such as `i32`.

The third parameter for `VideoP_A<N>` (line 27) is a queue-like data structure, `QueueAInit` (line 17), that codifies the order of usage of each binary channel inside a multiparty session channel. This is needed to preserve the causality, imposed by the global protocol. The queues for the other `SessionMpst` types are given in lines 21–24. For instance, the queue for the `ChoiceA:Video` branch of the protocol is `QueueAVideo`. Note that, according to the protocol, `A` first has to receive a `VideoRequest` message from `C`, and then it has to forward that message to `B` Hence, swapping of lines 17 and 18 from Fig. 2 is a protocol violation error. We can detect such violations since the queue for the type `ChoiceA::Video`, `QueueAVideo` (line 21), is specified as `RoleAtoC<RoleAtoB ...>`, which codifies that first the channel for `C` and then the channel for `B` should be used. Note that none of the defined queues is recursive. Recursion is implicitly specified on binary types, while each queue is related to a `SessionMpst` type.

Distributed Execution Environment. The default transport of `mpst-rust` is the built-in Rust communication channels (crossbeam_channel). Also, to test our example in a more realistic distributed environment, we have also connected each process through MQTT (MQ Telemetry Transport) [7]. MQTT is a messaging middleware for exchanging messages between devices, predominantly used in IoT networks. At the start of the protocol, each process connects to a public MQTT channel, and a session is established. Therefore, we have mapped binary channels to MQTT sockets, in addition to the built-in Rust channels.

3 Design and Implementation of `mpst-rust`

Multiparty Channels as an Ordered Tuple of Binary Channels. The main idea of the design of our framework is that a multiparty session can be realised with two ingredients: (1) a list of separate binary sessions (one session for each pair of participants) and (2) a queue that imposes the ordering between the binary channels. Listing 2 (lines 2–3) shows the implementation of a multiparty channel in a 3-party protocol. The `SessionMpst` structure holds two fields, `session1` and `session2`, that are of a binary session type. For an illustration purpose, we show only the implementation of a multiparty channel for three processes. The same approach can be generalised, using our code generation tool, to any number of communicating processes. For example, in case of a protocol with four roles, each multiparty session will have four fields – a field for the binary session between each pair of participants and a field for the queue.

The order of usage of the binary channels of a `SessionMpst` object is stored inside the `queue` field. For instance, the behaviour that role `A` has to communicate first with role `B`, then with a role `C`, and then the session ends can be specified using a queue of type `RoleAtoB<RoleAtoC<RoleEnd>>`. Note that all queue types, such as `RoleAtoB`, `RoleAtoC`, are generated.

MPST Communication Primitives as Wrappers of Binary Channels. As explained in Sect. 2, programming with `mpst-rust` relies on communication primitives, such as `send_mpst_a_to_b`, that have the sender and receiver roles baked into their name. To ensure that the binary channels are used as specified by the global protocol, each communication function is parametric on a generic quadruple type `<T, S1, S2, R>` where `T` is a payload type, `S1` and `S2` are binary session types and `R` is a type for a queue (MPST-queue type) that imposes the order in which the binary sessions inside a multiparty session must be used.

```
 1  // Basic structure for MPST
 2  pub struct SessionMpst< S1: Session, S2: Session, R: Role> {
 3   pub session1: S1, pub session2: S, pub queue: R }

 4  // Implementation of a communication function from the mpst-rust library
 5  pub fn send_mpst_a_to_b<T, S1, S2, R>(x: T,
 6   s: SessionMpst<Send<T, S1>, S2, RoleAtoB<R>>,) -> SessionMpst<S1, S2, R>
 7   where T: ..., S1: Session, S2: Session, R: Role,
 8   { let new_session = send(x, s.session1);
 9      let new_queue = next_a_to_b(s.queue); ... }

 9  /// Offer a choice at A from C wrapped in an ‘enum‘
10  #[macro_export]
11  macro_rules! offer_mpst_a_to_c {($session ... => {
12      let (l, s) = recv_mpst_a_to_c($session)?; // receive a label l
13      cancel(s); // cancel the existing binary channels on s
14      match l { pat_i=>invoke...} // Call the associated function
15          // with a new SessionMpst
16  }
17
18  #[macro_export]
19  macro_rules! choose_mpst_c_to_all(s ...) {
20      ...// test for the choice condition ...and get the label l
21      let s = send_mpst_c_to_a(s, l); // send the label to A
22      let s = send_mpst_c_to_b(s, l), // send the label to B
23      cancel(s); // cancel the existing binary channels on s
24      // return new SessionMpst channel
25      ...
26  }
```

Listing 2. MPST Rust communication primitives

Listing 2 (lines 5–9) shows the implementation for `send_mpst_a_to_b()`. As clear from the type parameters, the client of the function should supply a MPST-queue type `RoleAtoB<R>`. The binary session type `S1` should be encapsulated in a `Send<T, S1>`. The body of the function sends the message of type `T` on the binary channel stored in the first field, `session1` (corresponding to the binary session with role `B`), of the multiparty session `s`. Since the communication is on a binary channel, we reuse the binary `send` primitive from [9].

External and internal choices are implemented as macros that require an argument of type `SessionMpst`. The implementation of `offer_mpst_a_to_c` is given in lines 11–14. In essence, a choice is implemented as a broadcast from one role to the others. In our usecase, the active role that makes the choice is `C`. Hence, the macro `offer_mpst_a_to_c` explicitly performs a receive (`recv_mpst_a_to_c(s)`) on the session channel `s`. The received value is pattern matched and passed to any of the functions given as arguments to `offer_mpst_a_to_c`. Similarly, `choose_mpst_c_to_all` in lines 19–26 is a macro that performs a select operation. The active role `C` sends the selected label to all roles in the protocol. In our particular example, `C` sends the selected label `l` to `A` and `B`.

Discussions. Our implementation, although intuitive, does not resolve the inherent conflict between Rust, which is *affine*, and session types, which are *linear*. The implementation suffers from the same drawback as [9]. However, the MPST methodology is a step forward in terms of usability. Differently than the Rust local types which can get convoluted, the syntax of global protocols is user-friendly and readable. Developers can use the global protocol as guidance, and hence avoid errors such as prematurely ending of a session. Moreover, as observed in Kokke's library [9], most of the errors are caused by misuse of methods and functions. Since we are code-generating the local types, the chance of misspelling is significantly reduced. Another viable option for our framework is to take the *bottom-up* approach: to check directly whether a set of manually-written Rust local types satisfy *safety*/*liveness* properties by a model checker [14] or the *multiparty compatibility* (a property which guarantees deadlock-freedom of communicating automata, which are equivalent to local session types) [2,11].

4 Related and Future Work

The Rust library in [8] implements binary session types, following [4]. It checks at compile-time that the behaviours of two endpoint processes are *dual*, i.e the processes are compatible. The library in [9], based on the EGV calculus by Fowler *et al.* [3], provides constructs for writing and checking binary session types, and additionally supports exception handling constructs. We build on top of the library in [9] since it offers several improvements in comparison to [8]. Most importantly, the treatment of closing a channel prematurely in [8] may lead to memory leaks. Both libraries suffer from a well-known limitation of binary session types[3]. Notably, since deadlock-freedom is ensured only inside a session, a Rust endpoint process, that communicates with more than one other process, is prone to deadlocks and communication errors. Our framework solves that limitation by expanding the scope of a session to multiple participants.

Our proposed design follows the methodology given by [6], which generates Java communicating APIs from Scribble. This, and other multiparty session types implementations, exploit the equivalence between local session types and communicating automata to generate session types APIs for mainstream programming languages (e.g., Java [6,10], Go [1], F# [13]). Each state from state automata is implemented as a class, or in the case of [10], as a type state. To ensure safety, state automata have to be derived from the same global specification. All of the works in this category use the Scribble toolchain to generate the state classes from a global specification and detect linearity violations *at runtime*. This paper proposes the generation of protocol-specific APIs, which promotes type checking of protocols *at compile-time*. This is done by projecting the endpoints' state space in those protocols to groups of channel types in the desired language. In the future, we plan to implement the bottom-up approach, in addition to the top-down approach outlined in this paper, as to compare their productivity and scalability.

Acknowledgement. The work has been partially supported by the following funding schemes VeTSS, EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1 and, EP/T014709/1.

[3] https://github.com/Munksgaard/session-types/issues/62.

References

1. Castro, D., Hu, R., Jongmans, S.S., Ng, N., Yoshida, N.: Distributed programming using role parametric session types in Go. In: 46th ACM SIGPLAN Symposium on Principles of Programming Languages, vol. 3, pp. 29:1–29:30. ACM (2019)
2. Deniélou, P.-M., Yoshida, N.: Multiparty compatibility in communicating automata: characterisation and synthesis of global session types. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7966, pp. 174–186. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39212-2_18
3. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. Proc. ACM Program. Lang. **3**(POPL), 28:1–28:29 (2019). https://doi.org/10.1145/3290341
4. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
5. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. POPL **43**(1), 273–284 (2008)
6. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
7. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: MQTT-S-a publish/subscribe protocol for wireless sensor networks. In: 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops, COMSWARE 2008, pp. 791–798. IEEE (2008)
8. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for Rust. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, pp. 13–22. ACM (2015). https://doi.org/10.1145/2808098.2808100
9. Kokke, W.: Rusty variation: deadlock-free sessions with failure in Rust. In: Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20–21 June 2019, pp. 48–60 (2019). https://doi.org/10.4204/EPTCS.304.4
10. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo. In: PPDP, pp. 146–159 (2016). https://doi.org/10.1145/2967973.2968595
11. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 97–117. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_6
12. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for Rust. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, pp. 13–22. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2808098.2808100. ISBN 9781450338103
13. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: 31st European Conference on Object-Oriented Programming. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl (2017)
14. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. In: 46th ACM SIGPLAN Symposium on Principles of Programming Languages, vol. 3, pp. 30:1–30:29. ACM (2019)

15. Klabnik, S., Nichols, C.: The Rust Programming Language. 1.35.0 edn. (2019). https://doc.rust-lang.org/1.35.0/book/. Contributions from the Rust Community
16. Tu, T., Liu, X., Song, L., Zhang, Y.: Understanding real-world concurrency bugs in Go. In: ASPLOS, pp. 865–878. ACM (2019)
17. Yoshida, N., Gheri, L.: A very gentle introduction to multiparty session types. In: Hung, D.V., D'Souza, M. (eds.) ICDCIT 2020. LNCS, vol. 11969, pp. 73–93. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-36987-3_5
18. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 22–41. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05119-2_3