Safe Session-Based Asynchronous Coordination in Rust

Zak Cutner[0000−0001−7180−4530] and Nobuko Yoshida[0000−0002−3925−8557]

Imperial College London, London, UK

Abstract. Rust is a popular systems language focused on performance and reliability, with an emphasis on providing "fearless concurrency". *Message passing* has become a widely-used pattern by Rust developers although the potential for communication errors leaves developing safe and concurrent applications an unsolved challenge. In this ongoing work, we use *multiparty session types* to provide safety guarantees such as deadlock-freedom by coordinating message-passing processes. In contrast to previous contributions [22,21,20], our implementation targets *asynchronous* applications using `async/await` code in Rust. Specifically, we incorporate *asynchronous subtyping* theory, which allows program optimisation through reordering input and output actions. We evaluate our ideas by developing several representative use cases from the literature and by taking microbenchmarks. We discuss our plans to support full API generation integrating asynchronous optimisations.

Keywords: Rust · Asynchronous communication · Deadlock-freedom · Session types

1 Introduction

Rust is a statically typed language designed for systems software development. It is rapidly growing in popularity and has been voted "most loved language" over five years of surveys by Stack Overflow [12]. Rust aims to offer the safety of a high-level language without compromising on the performance enjoyed by low-level languages. *Message passing* over *typed channels* is widely used in concurrent Rust applications, whereby (low-level) threads or (high-level) actors communicate efficiently and safely by sending each other messages containing data.

This paper proposes a new implementation framework (Rumpsteak) for efficiently coordinating concurrent processes using *asynchronous* message-passing communication in Rust based on *multiparty session types* (MPST) [31,16,17]. MPST coordinate interactions through *linearly typed channels*, where each channel must be used exactly once, ensuring *protocol compliance* without deadlocks or communication mismatches. Rust's affine type system is particularly well-suited to MPST by statically guaranteeing a linear usage of session channels.

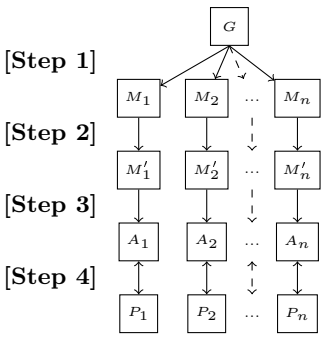
Previous implementations based on session types for Rust [22,21,20] operate under a *synchronous model*—that is upon attempting to receive a message, a thread is *blocked* until the operation has completed. Although simple, this

model can cause significant computational resources to be wasted while a thread is blocked and, moreover, many systems are inherently not synchronous. An *asynchronous model* instead makes no assumptions about how long an operation will take. After beginning to receive a message, a process can continue with its execution and be notified when the operation is complete. In practice, Rust supports the monadic `async`/`await` syntax to perform asynchronous operations. Functions that are asynchronous are annotated with `async`, causing them to return *futures*; and `await` is attached to futures, denoting that execution should continue elsewhere until the future is completed. Unfortunately, as shown in the Rust Survey 2020 [29], "`async`" and "concurrency" are ranked as the 5th and 7th most "tricky" or "very difficult" features among Rust programmers.

To improve both the safety and efficiency of communications in Rust, our work provides a Rust MPST toolchain (Rumpsteak), which supports asynchronous execution. We focus on two key challenges: **(C1)** how to correctly integrate MPST with Rust's `async`/`await` syntax, preserving safety and deadlock-freedom; and **(C2)** how to improve performance by using asynchronous execution. For **(C1)**, we develop a set of `async`/`await` primitives to build up MPST (see **§ 2**); and, for **(C2)**, we evaluate the efficiency of our primitives using microbenchmarks and develop several representative examples from the literature [10,24] with asynchronous communication optimisations (see **§ 3**). Finally, we discuss design choices for integration with advanced MPST theories, such as *asynchronous subtyping* [14] and *asynchronous multiparty compatibility* [24] to maximise communication speed-up, while still preserving safety between asynchronous components in Rust (see **§ 4**). We include further examples, source code and benchmarks in our repository [2] and the full version [11].

2 Overview

Workflow. Rumpsteak uses the *top-down* approach to ensure *correctness by design*. In **[Step 1]** we write a *global type* G to describe the interactions between all roles, and project it onto each role to obtain an endpoint finite state machine (EFSM) M_i; in **[Step 2]** we optimise each M_i to obtain M_i'; in **[Step 3]** we generate an API A_i from each M_i'; and in **[Step 4]** we use each A_i to create an asynchronous Rust process P_i. The group of processes $P_1 ... P_n$ created in this way are free from communication errors such as deadlocks.

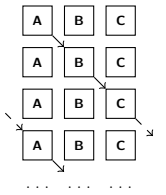


Projection. For **[Step 1]**, Rumpsteak uses νScr [3]: a new lightweight and extensible Scribble toolchain implemented in OCaml. The Scribble language [27,32] is widely used to describe multiparty protocols, agnostic to target languages. For illustration, we use a ring protocol extended with choice (`ring-choice` [11]),

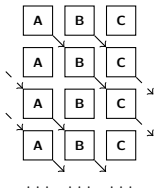
$$G = \mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \left\{ add(\texttt{i32}).\mathbf{B} \to \mathbf{C} : \left\{ \begin{array}{l} add(\texttt{i32}).\mathbf{C} \to \mathbf{A} : \{add(\texttt{i32}).\mathbf{t}\} \\ sub(\texttt{i32}).\mathbf{C} \to \mathbf{A} : \{sub(\texttt{i32}).\mathbf{t}\} \end{array} \right\} \right\}$$

Fig. 1: Global type for the `ring-choice` protocol

whose global type G is given in Fig. 1. Role **B** chooses between sending an *add* and *sub* message to role **C**, which must in turn send the same label to role **A**.



I (projected interactions)

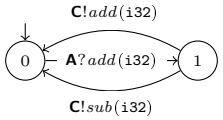


I' (optimised interactions)

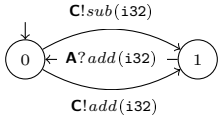
Asynchronous Optimisation. Since G is *synchronous*, naïvely projecting it onto **B** produces an overly synchronised EFSM $M_{\mathbf{B}}$[1]. If **A** is slow to send its value to **B** then the entire interaction is blocked (as shown in I). Instead, assuming each process begins with its own initial value, **B** could send its value to **C** in the meantime, allowing **C** to begin its next iteration (as shown in I').

Therefore, in **[Step 2]**, we transform $M_{\mathbf{B}}$ into the more optimal $M'_{\mathbf{B}}$. Importantly, we ensure that **(1)** no data dependencies exist between interactions, allowing their order to be changed; and **(2)** $M'_{\mathbf{B}}$ is an *asynchronous subtype* [14] of $M_{\mathbf{B}}$, allowing it to *safely* be used as a substitution while preserving deadlock-freedom. Presently, these steps are performed manually (see § 4).

Code Generation. Rumpsteak includes a code generator to produce an API in **[Step 3]**. Listing 1 shows the API $A_{\mathbf{B}}$ corresponding to the EFSM $M'_{\mathbf{B}}$, from which we have elided other participants. To ensure that our API remains readable by developers and to eliminate extensive boilerplate code, we make use of Rust procedural macros [28]. By decorating types with `#[...]`, these macros perform additional compile-time code generation. For each role, we generate a struct storing its communication channels with other roles. For example, **B** (line 3) contains unidirectional channels from **A** and to **C** as per the protocol. We use `#[derive(Role)]` to retrieve channels from the struct.



$M_{\mathbf{B}}$ (projected)



$M'_{\mathbf{B}}$ (optimised)

Following the approach of [22], we build a set of *generic primitives* to construct a simple API—reducing the amount of generated code and avoiding arbitrarily named types. For instance, the `Receive` primitive (line 22) takes a role, label and continuation as generic parameters. For readability, we elide two additional parameters used to store channels at runtime with `#[session]`.

[1] We use session type syntax [31] where ! and ? denote send and receive respectively.

```
1  #[derive(Role)]
2  #[message(Label)]
3  struct B {
4    #[route(A)] a: Receiver,
5    #[route(C)] c: Sender,
6  }
7
8  #[derive(Message)]
9  enum Label {
10   Add(Add),
11   Sub(Sub),
12 }
13
14 struct Add(i32);
15 struct Sub(i32);
16
17 #[session]
18 type RingB = Select<C, RingBChoice>;
19
20 #[session]
21 enum RingBChoice {
22   Add(Add, Receive<A, Add, RingB>),
23   Sub(Sub, Receive<A, Add, RingB>),
24 }
```

Listing 1: Rust session type API for $M'_{\mathbf{B}}$ ($A_{\mathbf{B}}$)

```
1  async fn ring_b(
2    role: &mut B,
3    mut input: i32,
4  ) -> Result<Infallible> {
5    try_session(
6      role,
7      |mut s: RingB<'_, _>| async {
8        loop {
9          let x = input * 2;
10         s = if x > 0 {
11           let s = s.select(Add(x)).await?;
12           let (Add(y), s) = s.receive().await?;
13           input = y + x;
14           s
15         } else {
16           let s = s.select(Sub(x)).await?;
17           let (Add(y), s) = s.receive().await?;
18           input = y - x;
19           s
20         };
21       }
22     },
23   ).await
24 }
```

Listing 2: Possible Rust implementation for process $\mathbf{B}$ ($P_{\mathbf{B}}$) using $A_{\mathbf{B}}$

Each choice generates an enum, as seen in `RingBChoice` (line 21), allowing processes to pattern match when branching to determine which label was received. Methods allowing the enum to be used with `Branch` or `Select` primitives are also generated with `#[session]`. An enum is required since Rust's lack of variadic generics means choice cannot be easily implemented as a primitive. We show how the `RingBChoice` type can be used with selection in the `Ring` type (line 18).

Our API requires only one session type for each role, internally sending a `Label` enum (line 9) over reusable channels. We create a type for each label (lines 14 and 15) and use `#[derive(Message)]` to generate methods for converting to and from the `Label` enum. In contrast, [22] requires a tuple of binary sessions for each role and communicates using typed, one-shot channels. Our approach is simpler, requiring fewer definitions, and also more performant (see **§ 3**).

Process Implementation. Using the API $A_{\mathbf{B}}$, we suggest a possible implementation of the process $P_{\mathbf{B}}$, shown in Listing 2, for **[Step 4]**. Linear usage of channels is checked by Rust's *affine type system* to prevent channels from being used multiple times. When a primitive is executed, it consumes itself, preventing reuse, and returns its continuation. While [21] and [22] use compiler hints to warn the programmer when a session is discarded without use, we ensure this *statically* by harnessing the type checker. Developers are prevented from constructing primitives directly using visibility modifiers and must instead use `try_session` (line 5). Its closure argument accepts the input session type and returns the terminal type `End`. If a session is discarded, breaking linearity, then the developer will have no `End` to return and the type checker will complain. Even

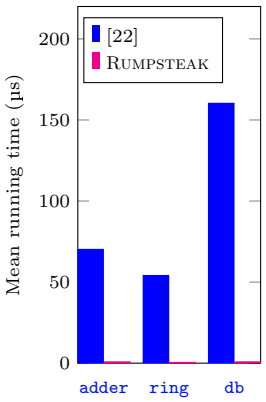


Fig. 2: Comparison of Rumpsteak and [22]

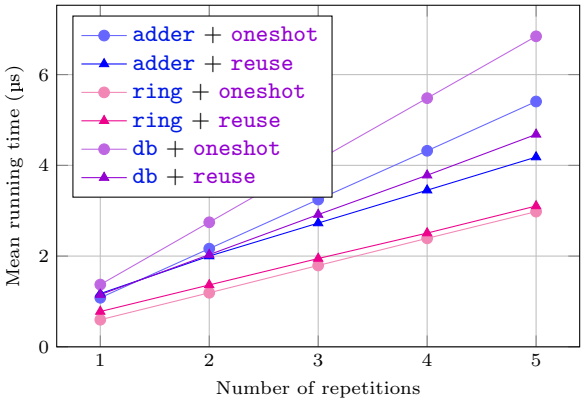


Fig. 3: Comparison of `oneshot` and `reuse` under asynchronous execution

so, we can implement processes with infinitely recursive types (containing no `End`) such as `RingB`. We use an infinite loop (line 8) which is assigned `Infallible`: Rust's never (or bottom) type. `Infallible` can be implicitly cast to any other type, including `End`, allowing the closure to pass the type checker as before.

We allow roles to be reused across sessions since the channels they contain can be expensive to create. Crucially, to prevent communication mismatches between different sessions, `try_session` takes a *mutable* reference to the role. The same role, therefore, cannot be used multiple times at once because Rust's borrow checker enforces this requirement for mutable references.

3 Evaluation

Microbenchmarks. We investigate Rumpsteak's performance, comparing it with the most recent related work [22]. We introduce three protocols from [11]:

- (`adder`) an adder protocol between three participants;

$$G = \mathbf{A} \to \mathbf{B} : \{add(\texttt{i32}).\mathbf{B} \to \mathbf{A} : \{add(\texttt{i32}).\mathbf{A} \to \mathbf{C} : \{add(\texttt{i32}).\\ \mathbf{B} \to \mathbf{C} : \{add(\texttt{i32}).\mathbf{C} \to \mathbf{A} : \{sum(\texttt{i32}).\mathbf{C} \to \mathbf{B} : \{sum(\texttt{i32}).\texttt{end}\}\}\}\}\}$$

- (`ring`) a simpler version of `ring-choice`; and

$$G = \mathbf{A} \to \mathbf{B} : \{value(\texttt{i32}).\mathbf{B} \to \mathbf{C} : \{value(\texttt{i32}).\mathbf{C} \to \mathbf{A} : \{value(\texttt{i32}).\texttt{end}\}\}\}$$

- (`db`) a double buffering protocol [19] between source **S**, kernel **K** and sink **T**.

$$G = \mathbf{K} \to \mathbf{S} : \{ready.\mathbf{S} \to \mathbf{K} : \{copy(\texttt{i32}).\mathbf{T} \to \mathbf{K} : \{ready.\\ \mathbf{K} \to \mathbf{T} : \{copy(\texttt{i32}).\mathbf{K} \to \mathbf{S} : \{ready.\mathbf{S} \to \mathbf{K} : \{copy(\texttt{i32}).\\ \mathbf{T} \to \mathbf{K} : \{ready.\mathbf{K} \to \mathbf{T} : \{copy(\texttt{i32}).\texttt{end}\}\}\}\}\}\}\}$$

Only terminating protocols are used so that we can practically measure their running times. Most previous session type implementations in Rust [21,20] support only *binary* protocols. Our contribution and benchmarks instead target *multiparty* protocols, therefore we compare Rumpsteak only against [22] which has a similar scope. Since [22] is built upon [21], we expect both to have similar performance for binary protocols. We execute all benchmarks using an 8-core Intel® Core™ i7-7700K CPU @ 4.20 GHz with hyperthreading, 16GB RAM, Ubuntu 20.04.2 LTS and Rust 1.51.0. We use version 0.3.4 of the Criterion.rs library [15] to perform microbenchmarking and a *single-threaded* asynchronous runtime from version 1.5.0 of the Tokio library [30].

Our first benchmark (Fig. 2) performs a direct comparison between Rumpsteak and [22] for all three protocols. It shows that Rumpsteak can run these protocols around 50-150 times faster. We attribute this to our approach of using asynchronous execution. Asynchronous tasks are significantly more lightweight than kernel threads and so incur much lower overheads. We note that blocking operations do not contribute to weaker synchronous performance as we benchmark with significantly more cores than the number of roles.

As discussed previously (see **§ 2**), Rumpsteak uses reusable channels in contrast to one-shot channels used by [22]. To compare both approaches fairly, Fig. 3 benchmarks Rumpsteak (`reuse`) against a subset of [22] ported to use asynchronous execution (`oneshot`). We simulate a longer protocol by reusing the same channels for a parameterised number of repetitions, although one-shot channels, by design, cannot be reused. In `adder` and `ring`, `oneshot` performs better than `reuse` for a single iteration. However, as the number of repetitions increases, constructing a growing number of one-shot channels quickly outweighs the one-time instantiation penalty of reusable channels. By the second iteration, `reuse` overtakes the performance of `oneshot` in `adder` and `db`. Only for `ring` (which contains the least number of exchanges) is `oneshot` still faster after five iterations, although the gradients suggest that `reuse` will eventually overtake. We conclude that `reuse` is more efficient than `oneshot` in all but the shortest protocols.

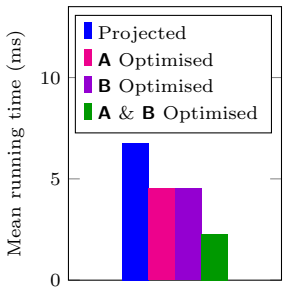


Fig. 4: Comparison of `ring` optimisations

We explore an asynchronous optimisation to `ring` in Fig. 4 by swapping **A**/**B**'s input and output actions. We also insert artificial 1ms communication delays to simulate a more realistic scenario. We observe a significant performance improvement by applying the optimisation to either **A** or **B**. Moreover, the effect is compounded by optimising both participants at once, resulting in a ⅔ speed-up.

Fig. 5 shows an asynchronous optimisation to `db` whereby **K** sends **S** both *ready* messages at once, further discussed in the full version [11]. We insert similar artificial communication delays between **S** and **K** for our experimental setup. Interestingly, this optimisation causes duality between **S** and **K** to break. Since [22] uses a tuple of binary session types for each role, it is crucially unable to express this optimi-

sation. Therefore, we further propose a weaker optimisation for [22] by sending the second *ready* message only after K has received the first *copy* message to preserve duality. Fig. 5 shows that while this weaker optimisation has little effect on performance, the original and stronger optimisation which is expressable by Rumpsteak results in around a 25% improvement.

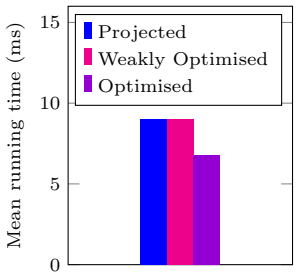


Fig. 5: Comparison of db optimisations

Expressiveness. We further illustrate the expressiveness of Rumpsteak compared with [22] in Fig. 6. We implement several examples of protocols from the literature using Rumpsteak. For each example, we detail its key features, particularly if it makes use of asynchronous optimisations, and whether we can also express the protocol using [22].

Our results demonstrate that [22] is less expressive than Rumpsteak for asynchronously-optimised protocols since its workflow does not include an optimisation step. Some optimisations, which are marked with ◖, can nevertheless be expressed in [22] by implementing them directly using its endpoint API. However, this method does not benefit from using the workflow in [22]. Even then, as discussed for db, [22] uses a tuple of binary session types for each role and therefore any optimisation must not break duality between each pair of participants. Unfortunately, this prohibits it from performing most asynchronous optimisations, even in this more limited way. In contrast, Rumpsteak enjoys complete flexibility to perform more complex optimisations in a wide-ranging number of examples from the literature.

Conclusion. By using asynchronous execution, Rumpsteak is around two orders of magnitude faster than [22], and this benefit is even greater in longer-running protocols due to our use of reusable channels. We observe the need for asynchronous optimisation by demonstrating several significant performance improvements and show that, in several cases, Rumpsteak can express stronger and more valuable optimisations than are expressable in [22].

4 Related and Future Work

There are a vast number of studies on session types, some of which are implemented in programming languages [4] and tools [13]. A code generation toolchain takes a protocol description and produces well-typed APIs, conforming to that protocol. Several implementations use an EFSM-based approach to generate APIs from Scribble [27,32,3] for target programming languages such as Java [18], F# [26], Go [9], F⋆ [33] and TypeScript [25]. We closely compared with previous work on API generation in Rust from MPST protocols [22] (see detailed comparisons with [21] and [20] in [22]). We justify our work is **(1)** robust, using

Protocol	Features				Expressable	
	C	R	IR	AO	[22]	RUMPSTEAK
Two Adder [3]	●	●			●	●
`adder` [11]					●	●
`ring` [10,11]		●[a]	●[a]		●	●
Optimised `ring` [10,11]		●[a]	●[a]	●	◖	●
`ring-choice` [10,11]	●	●	●		●	●
Optimised `ring-choice` [10,11]	●	●	●	●	◖	●
`db` [10,11]		●[a]	●[a]		●	●
Optimised `db` [10,19,11]		●[a]	●[a]	●		●
Alternating Bit [24,1]	●	●	●	●		●
Elevator[b] [24,5]	●	●	●	●		●
FFT [10]					●	●

C Contains choice R Recursive IR Infinitely recursive AO Uses asynchronous optimisations

[a] Although non-recursive, we can easily extend the protocol to make it recursive.
[b] We use the communicating session automata variation from [24].

Fig. 6: Expressiveness of [22] and RUMPSTEAK

affine typing, while providing a simpler API (see **§ 2**) and **(2)** faster and more expressive by using `async`/`await` and reusable channels (see **§ 3**). Here, our aim is ensuring *correctness/safety by construction*, maximising *asynchrony* for gaining *efficiency* of message passing in Rust applications.

Our main remaining challenge is how to validate the well-formedness of a set of optimised EFSMs, i.e. $\{M_i'\}_{i\in I}$ generated in **[Step 2]** of the workflow presented in **§ 2**. One possible approach is the use of *multiparty asynchronous subtyping* [14] to validate $M_i' \leqslant M_i$ for each participant. Asynchronous session subtyping was shown to be undecidable, even for binary sessions [23,8], hence, in general, checking $M_i' \leqslant M_i$ is undecidable. Various limited classes of session types for which $M_i' \leqslant M_i$ is decidable [6,7,23,10] are proposed but not applicable to our use cases since **(1)** the relations in [6,8,23] are *binary* and the same limitations do not work for multiparty; and **(2)** the relation in [10, Def. 6.1] does not handle subtyping across unrolling recursions, e.g. the relation is inapplicable to the double buffering algorithm [19] (see [10, Remark 8.1]). Hence, we need to find non-trivial decidable approximations of our multiparty asynchronous subtyping relation. The second approach is to use *k-multiparty compatibility* developed in [24] to analyse a whole set of $\{M_i'\}_{i\in I}$. We investigate both options and report our findings at the conference presentation.

Acknowledgements. We thank Nicolas Lagaillardie and Fangyi Zhou for their helpful comments and suggestions. The work is supported by EPSRC, grants EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T014709/1, and EP/V000462/1 and by NCSS/EPSRC VeTSS.

References

1. Introduction to Protocol Engineering. http://cs.uccs.edu/~cs522/pe/pe.htm, [Accessed 19 February 2021]
2. Rumpsteak. https://github.com/zakcutner/rumpsteak
3. νScr. https://github.com/nuscr/nuscr
4. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P.M., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral Types in Programming Languages. Foundations and Trends Programming Languages **3**(2–3), 95–230 (2016)
5. Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In: Computer Aided Verification. pp. 372–391. CAV, Springer (2018)
6. Bravetti, M., Carbone, M., Lange, J., Yoshida, N., Zavattaro, G.: A Sound Algorithm for Asynchronous Session Subtyping **140**, 38:1–38:16 (2019)
7. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of Asynchronous Session Subtyping. Information and Computation **256**, 300–320 (2017)
8. Bravetti, M., Carbone, M., Zavattaro, G.: On the Boundary between Decidability and Undecidability of Asynchronous Session Subtyping. Theoretical Computer Science **722**, 19–51 (2018)
9. Castro, D., Hu, R., Jongmans, S.S., Ng, N., Yoshida, N.: Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures. Proceedings of the ACM on Programming Languages **3**(POPL), 29:1–29:30 (2019)
10. Castro-Perez, D., Yoshida, N.: CAMP: Cost-Aware Multiparty Session Protocol. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–30 (2020)
11. Cutner, Z., Yoshida, N.: Safe Session-Based Asynchronous Coordination in Rust. https://github.com/zakcutner/coordination-2021
12. Donovan, R.: Why the Developers Who Use Rust Love It so Much. https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/ (2020), [Accessed 31 January 2021]
13. Gay, S., Ravara, A.: Behavioural Types: from Theory to Tools. River Publisher (2017)
14. Ghilezan, S., Pantovic, J., Prokic, I., Scalas, A., Yoshida, N.: Precise Subtyping for Asynchronous Multiparty Sessions. In: Proceedings of the ACM on Programming Languages. POPL, vol. 5, pp. 16:1–16:28. ACM (2021)
15. Heisler, B.: Criterion.rs. https://github.com/bheisler/criterion.rs
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: Proceedings of the ACM on Programming Languages. pp. 273–284. POPL, ACM (2008)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. JACM **63**, 1–67 (2016)
18. Hu, R., Yoshida, N.: Hybrid Session Verification through Endpoint API Generation. In: FASE. LNCS, vol. 9633, pp. 401–418. Springer (2016)
19. Huang, H., Pillai, P., Shin, K.G.: Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems. In: 2002 USENIX Annual Technical Conference (USENIX ATC 02). USENIX Association (2002)

20. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session Types for Rust. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming. p. 13–22. WGP, ACM (2015)
21. Kokke, W.: Rusty Variation: Deadlock-free Sessions with Failure in Rust. Electronic Proceedings in Theoretical Computer Science **304**, 48–60 (2019)
22. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing Multiparty Session Types in Rust. In: COORDINATION. LNCS, vol. 12134, pp. 127–136. Springer (2020)
23. Lange, J., Yoshida, N.: On the Undecidability of Asynchronous Session Subtyping. In: FoSSaCS. LNCS, vol. 10203, pp. 441–457 (2017)
24. Lange, J., Yoshida, N.: Verifying Asynchronous Interactions via Communicating Session Automata. In: CAV. LNCS, vol. 11561, pp. 117–97. Springer (2019)
25. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types. In: Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction. p. 94–106. CC, ACM (2021)
26. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A Session Type Provider: Compile-time API Generation of Distributed Protocols with Refinements in F#. In: 27th International Conference on Compiler Construction. pp. 128–138. CC, ACM (2018)
27. Scribble Authors: Scribble: Describing Multi Party Protocols. http://www.scribble.org/ (2015)
28. The Rust Project Developers: Procedural Macros. https://doc.rust-lang.org/reference/procedural-macros.html
29. The Rust Survey Team: Rust Survey 2020 Results. https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html (2020), [Accessed 31 January 2021]
30. Tokio Contributors: Tokio. https://github.com/tokio-rs/tokio
31. Yoshida, N., Gheri, L.: A Very Gentle Introduction to Multiparty Session Types. In: 16th International Conference on Distributed Computing and Internet Technology. LNCS, vol. 11969, pp. 73–93. Springer (2020)
32. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble Protocol Language. In: 8th International Symposium on Trustworthy Global Computing. LNCS, vol. 8358, pp. 22–41. Springer (2013)
33. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically Verified Refinements for Multiparty Protocols. Proceedings of the ACM on Programming Languages **4**(OOPSLA) (2020)

A Three Adder Protocol

The three adder protocol (`adder`) is a simple protocol previously introduced in § **3**. It consists of three participants: **A**, **B** and **C**. **A** and **B** each pick an integer and send it to one another, then send the other's selection to **C**. **C** adds both integers and sends the result to both **A** and **B**. We show the Scribble protocol in Listing 3 and a Rust implementation in Listings 4 and 5.

```
global protocol ThreeAdder(role A, role B, role C) {
  add(i32) from A to B; add(i32) from B to A;
  add(i32) from A to C; add(i32) from B to C;
  sum(i32) from C to A; sum(i32) from C to B;
}
```

Listing 3: Scribble representation of `adder`

```
#[derive(Roles)]
struct Roles(A, B, C);

#[derive(Role)]
#[message(Label)]
struct A(#[route(B)] Channel, #[route(C)] Channel);

#[derive(Role)]
#[message(Label)]
struct B(#[route(A)] Channel, #[route(C)] Channel);

#[derive(Role)]
#[message(Label)]
struct C(#[route(A)] Channel, #[route(B)] Channel);

#[derive(Message)]
enum Label {
  Add(Add),
  Sum(Sum),
}

struct Add(i32);
struct Sum(i32);

#[session]
type AdderA = Send<B, Add, Receive<B, Add, Send<C, Add, Receive<C, Sum, End>>>>;

#[session]
type AdderB = Receive<A, Add, Send<A, Add, Send<C, Add, Receive<C, Sum, End>>>>;

#[session]
type AdderC = Receive<A, Add, Receive<B, Add, Send<A, Sum, Send<B, Sum, End>>>>;
```

Listing 4: Rust session type API for `adder`

```rust
async fn adder_a(role: &mut A, x: i32) -> Result<i32> {
  try_session(role, |s: AdderA<'_, _>| async {
    let s = s.send(Add(x)).await?;
    let (Add(y), s) = s.receive().await?;
    let s = s.send(Add(y)).await?;
    let (Sum(z), s) = s.receive().await?;
    Ok((z, s))
  })
  .await
}

async fn adder_b(role: &mut B, x: i32) -> Result<()> {
  try_session(role, |s: AdderB<'_, _>| async {
    let (Add(y), s) = s.receive().await?;
    let s = s.send(Add(x)).await?;
    let s = s.send(Add(y)).await?;
    let (Sum(z), s) = s.receive().await?;
    Ok((z, s))
  })
  .await
}

async fn adder_c(role: &mut C) -> Result<()> {
  try_session(role, |s: AdderC<'_, _>| async {
    let (Add(x), s) = s.receive().await?;
    let (Add(y), s) = s.receive().await?;
    let z = x + y;
    let s = s.send(Sum(z)).await?;
    Ok(((), s.send(Sum(z)).await?))
  })
  .await
}
```

Listing 5: Possible Rust process implementations for `adder`

B Ring Protocol

The ring protocol [10], discussed in § **2**, allows three processes—**A**, **B** and **C**—arranged in a ring to perform a distributed computation. Each process begins with an initial input, performs a computation on its input and sends the result of the computation to the succeeding process in the ring. The protocol then repeats, using the value received from the preceding process as the new input.

We implement two variants of the ring protocol. `ring-choice` (see § **2**) is infinitely recursive and includes a choice at **B** while `ring` has no choice and runs only for a single iteration (see § **3**).

B.1 `ring-choice`

We show the Scribble protocol in Listing 6. In addition to the EFSMs and Rust implementation for **B** shown in § **2**, we show the EFSMs for **A** and **B** in Figs. 7 and 8 and a complete Rust implementation in Listings 7 and 8.

```
1  global protocol RingChoice(role A, role B, role C) {
2    add(i32) from A to B;
3    choice at B {
4      add(i32) from B to C;
5      add(i32) from C to A;
6      do Ring(A, B, C);
7    } or {
8      subtract(i32) from B to C;
9      subtract(i32) from C to A;
10     do Ring(A, B, C);
11   }
12 }
```

Listing 6: Scribble representation of `ring-choice`

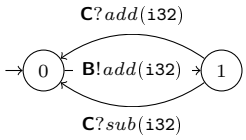


Fig. 7: $M_{\mathbf{A}}$ for `ring-choice`

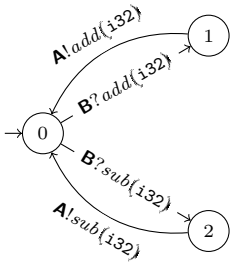


Fig. 8: $M_{\mathbf{C}}$ for `ring-choice`

```
#[derive(Roles)]
struct Roles(A, B, C);

#[derive(Role)]
#[message(Label)]
struct A(#[route(B)] Sender, #[route(C)] Receiver);

#[derive(Role)]
#[message(Label)]
struct B(#[route(A)] Receiver, #[route(C)] Sender);

#[derive(Role)]
#[message(Label)]
struct C(#[route(A)] Sender, #[route(B)] Receiver);

#[derive(Message)]
enum Label {
  Add(Add),
  Sub(Sub),
}

struct Add(i32);
struct Sub(i32);

#[session]
type RingA = Send<B, Add, Branch<C, RingAChoice>>;

#[session]
enum RingAChoice {
  Add(Add, RingA),
  Sub(Sub, RingA),
}

#[session]
type RingB = Select<C, RingBChoice>;

#[session]
enum RingBChoice {
  Add(Add, Receive<A, Add, RingB>),
  Sub(Sub, Receive<A, Add, RingB>),
}

#[session]
type RingC = Branch<B, RingCChoice>;

#[session]
enum RingCChoice {
  Add(Add, Send<A, Add, RingC>),
  Sub(Sub, Send<A, Sub, RingC>),
}
```

Listing 7: Rust session type API for `ring-choice`

```rust
async fn ring_a(role: &mut A, mut input: i32) -> Result<Infallible> {
  try_session(role, |mut s: RingA<'_, _>| async {
    loop {
      let x = input * 2;
      s = match s.send(Add(x)).await?.branch().await? {
        RingAChoice::Add(Add(y), s) => {
          input = x + y;
          s
        }
        RingAChoice::Sub(Sub(y), s) => {
          input = x - y;
          s
        }
      };
    }
  })
  .await
}

async fn ring_b(role: &mut B, mut input: i32) -> Result<Infallible> {
  try_session(role, |mut s: RingB<'_, _>| async {
    loop {
      let x = input * 2;
      s = if x > 0 {
        let s = s.select(Add(x)).await?;
        let (Add(y), s) = s.receive().await?;
        input = y + x;
        s
      } else {
        let s = s.select(Sub(x)).await?;
        let (Add(y), s) = s.receive().await?;
        input = y - x;
        s
      };
    }
  })
  .await
}

async fn ring_c(role: &mut C, mut input: i32) -> Result<Infallible> {
  try_session(role, |mut s: RingC<'_, _>| async {
    loop {
      let x = input * 2;
      s = match s.branch().await? {
        RingCChoice::Add(Add(y), s) => {
          let s = s.send(Add(x)).await?;
          input = x + y;
          s
        }
        RingCChoice::Sub(Sub(y), s) => {
          let s = s.send(Sub(x)).await?;
          input = x - y;
          s
        }
      };
    }
  })
  .await
}
```

Listing 8: Possible Rust process implementations for `ring-choice`

B.2 ring

We show the Scribble protocol in Listing 9 and a Rust implementation in Listings 10 and 11. As described in § **3**, we can perform an asynchronous optimisation to **B** and **C** by swapping their input and output actions.

```
global protocol Ring(role A, role B, role C) {
  value(i32) from A to B;
  value(i32) from B to C;
  value(i32) from C to A;
}
```

Listing 9: Scribble representation of ring

```
#[derive(Roles)]
struct Roles(A, B, C);

#[derive(Role)]
#[message(Value)]
struct A(#[route(B)] Sender, #[route(C)] Receiver);

#[derive(Role)]
#[message(Value)]
struct B(#[route(A)] Receiver, #[route(C)] Sender);

#[derive(Role)]
#[message(Value)]
struct C(#[route(A)] Sender, #[route(B)] Receiver);

#[derive(Message)]
struct Value(i32);

#[session]
type RingA = Send<B, Value, Receive<C, Value, End>>;

#[session]
type RingB = Send<C, Value, Receive<A, Value, End>>;

#[session]
type RingC = Send<A, Value, Receive<B, Value, End>>;
```

Listing 10: Rust session type API for ring

```
async fn ring_a(role: &mut A, x: i32) -> Result<i32> {
  try_session(role, |s: RingA<'_, _>| async {
    let s = s.send(Value(x)).await?;
    let (Value(y), s) = s.receive().await?;
    Ok((y, s))
  })
  .await
}

async fn ring_b(role: &mut B, x: i32) -> Result<()> {
  try_session(role, |s: RingB<'_, _>| async {
    let s = s.send(Value(x)).await?;
    let (Value(y), s) = s.receive().await?;
    Ok((y, s))
  })
  .await
}

async fn ring_c(role: &mut C, x: i32) -> Result<()> {
  try_session(role, |s: RingC<'_, _>| async {
    let s = s.send(Value(x)).await?;
    let (Value(y), s) = s.receive().await?;
    Ok((y, s))
  })
  .await
}
```

Listing 11: Possible Rust process implementations for `ring`

C Double Buffering Protocol

In high-performance computing, safely and efficiently sending data from a source to a sink running concurrently is a well-known problem. For example, many media applications require efficiently streaming data to or from external devices such as graphics or sound cards. When the sink is not ready to receive messages that the source has prepared this problem becomes challenging—blocking the source is not acceptable since this reduces the parallelism of the application. Instead, the use of buffers is a common technique used to improve performance. Rather than sending directly to the sink, the source puts values into a buffer. When the sink is ready to receive, it simply reads values from the buffer or waits for a value to be added if the buffer is empty. Importantly, putting values in the buffer allows the source to continue with other work rather than waiting for the sink to become ready to receive.

However, buffer implementations themselves must be safe to be read from and written to concurrently, requiring the use of locking. If both the source and sink are ready to exchange data at the same time, they will, unfortunately, become blocked at this locking stage as they transfer data to and from the buffer. Instead, the double buffering algorithm makes use of two buffers controlled by a kernel [19]. Both the source and sink alternate between using the first and second buffer such that they never use the same buffer at once. This conveniently sidesteps the issue of locking since the source and sink operate on different buffers so do not compete for the same lock.

Previously presented in § **3**, we use a variation of the double buffering protocol (`db`) from [10]. Our algorithm, consisting of a source **S**, a kernel **K** and a sink **T**, is modified to have a single iteration so that it can be practically benchmarked. Importantly, **K** communicates on behalf of both buffers allowing the implementations of **S** and **T** to be agnostic of how many buffers **K** holds.

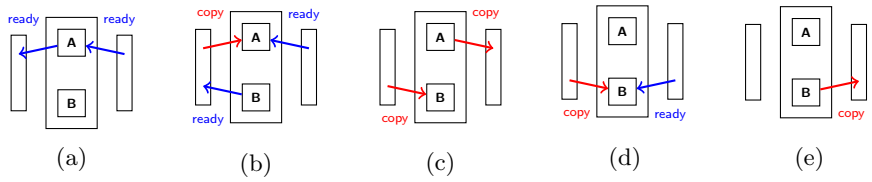


Fig. 9: Illustration of the interactions in `db`

We use Fig. 9 to illustrate the execution of the protocol and describe the interactions between participants.

(a) **K** notifies **S** that it is ready to receive a value.
(b) **S** copies its first value to **K** while **K** notifies **S** that it is ready for the second value. Meanwhile, **T** has notified **K** that it is ready to receive its first value.
(c) **K** copies the first value to **T** while **S** copies the second value to **K**.
(d) **T** notifies **K** that it is ready to receive the second value.
(e) **K** copies the second value to **T**.

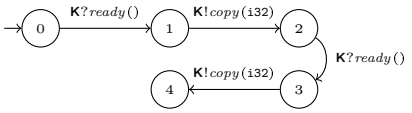
We could then extend the protocol, allowing additional values to be sent if we wished to facilitate more than a single iteration.

```
1  global protocol DoubleBuffering(role S, role K, role T) {
2    ready() from K to S;
3    copy(i32) from S to K;
4    ready() from T to K;
5    copy(i32) from K to T;
6    ready() from K to S;
7    copy(i32) from S to K;
8    ready() from T to K;
9    copy(i32) from K to T;
10 }
```

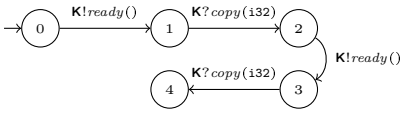
Listing 12: Scribble representation of `db`

In particular, we note the *asynchrony* of the protocol; for example, **S** sends its first value to **K** while **T** notifies **K** that it is ready to receive. In contrast, the Scribble representation of this protocol, presented in Listing 12, is synchronous; **S** copies to **K** (line 3) *before* **T** notifies **K** (line 4).

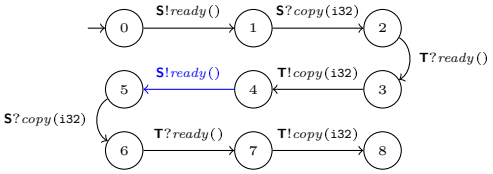
Therefore, projecting the Scribble protocol onto each role will produce the EFSMs $M_{\mathbf{S}}$, $M_{\mathbf{K}}$ and $M_{\mathbf{T}}$, shown in Fig. 10. However, as previously discussed in



(a) $M_{\mathbf{S}}$



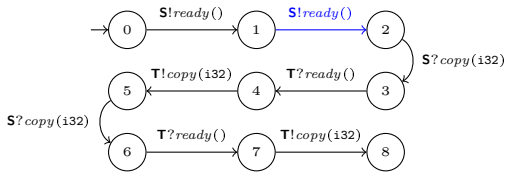
(b) $M_{\mathbf{T}}$



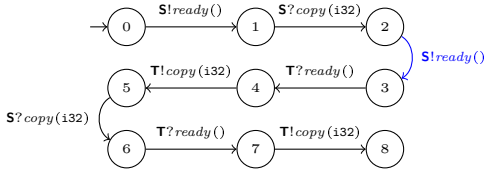
(c) $M_{\mathbf{K}}$ (projected)

Fig. 10: EFSMs for the double buffering protocol

§ **3**, we note that $M_{\mathbf{K}}$ is overly synchronised. It waits to send its second *ready* message to **S** (coloured in blue) until the first value has been copied to **T** although both buffers are in fact ready from the start.



(a) $M'_{\mathbf{K}}$ (optimised)



(b) $M''_{\mathbf{K}}$ (weakly optimised)

Fig. 11: Optimised EFSMs for the double buffering protocol

By performing the optimisation described in § **3**, we send the second *ready* message immediately to allow **S** to send both values before **K** has even communicated with **T**. We show this optimised EFSM, $M'_{\mathbf{K}}$, in Fig. 11a. Unfortunately, this optimisation is not possible with the one-shot approach; the binary session types between **S** and **K** no longer satisfy duality using $M_{\mathbf{S}}$ and $M'_{\mathbf{K}}$. For its sec-

ond interaction, **S** expects to send a *copy* message to **K**, but simultaneously **K** is expecting to send a *ready* message to **S**.

However, it is still possible to perform a weaker, albeit less performant (see § **3**) optimisation that does preserve duality. As shown in Fig. 11b, we can produce $M''_{\mathbf{K}}$ by sending the second *ready* message after the first value has been copied to **K**, which is still earlier than before. We show a Rust implementation of the protocol in Listings 13 and 14.

```
#[derive(Roles)]
struct Roles(S, K, T);

#[derive(Role)]
#[message(Label)]
struct S(#[route(K)] Channel, #[route(T)] Nil);

#[derive(Role)]
#[message(Label)]
struct K(#[route(S)] Channel, #[route(T)] Channel);

#[derive(Role)]
#[message(Label)]
struct T(#[route(S)] Nil, #[route(K)] Channel);

#[derive(Message)]
enum Label {
  Ready(Ready),
  Copy(Copy),
}

struct Ready;
struct Copy(i32);

#[session]
type Source = Receive<K, Ready, Send<K, Copy, Receive<K, Ready, Send<K, Copy, End>>>>;

#[session]
type Kernel = Send<S, Ready, Send<S, Ready, Receive<S, Copy, Receive<T, Ready, Send<T,
  Copy, Receive<S, Copy, Receive<T, Ready, Send<T, Copy, End>>>>>>>;

#[session]
type Sink = Send<K, Ready, Receive<K, Copy, Send<K, Ready, Receive<K, Copy, End>>>>;
```

Listing 13: Rust session type API for `db`

```rust
async fn source(role: &mut S, x: i32, y: i32) -> Result<()> {
  try_session(role, |s: Source<'_, _>| async {
    let (Ready, s) = s.receive().await?;
    let s = s.send(Copy(x)).await?;

    let (Ready, s) = s.receive().await?;
    let s = s.send(Copy(y)).await?;

    Ok(((), s))
  })
  .await
}

async fn kernel(role: &mut K) -> Result<()> {
  try_session(role, |s: Kernel<'_, _>| async {
    let s = s.send(Ready).await?;
    let s = s.send(Ready).await?;

    let (Copy(x), s) = s.receive().await?;
    let (Ready, s) = s.receive().await?;
    let s = s.send(Copy(x)).await?;

    let (Copy(y), s) = s.receive().await?;
    let (Ready, s) = s.receive().await?;
    let s = s.send(Copy(y)).await?;

    Ok(((), s))
  })
  .await
}

async fn sink(role: &mut T) -> Result<(i32, i32)> {
  try_session(role, |s: Sink<'_, _>| async {
    let s = s.send(Ready).await?;
    let (Copy(x), s) = s.receive().await?;

    let s = s.send(Ready).await?;
    let (Copy(y), s) = s.receive().await?;

    Ok(((x, y), s))
  })
  .await
}
```

Listing 14: Possible Rust process implementations for `db`