Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types (Artifact)

Nicolas Lagaillardie
Department of Computing, Imperial College London, UK

Rumyana Neykova
Department of Computer Science, Brunel University London, UK

Nobuko Yoshida
Department of Computing, Imperial College London, UK

Abstract

This artifact contains a version of `MultiCrusty`, a Rust library designed for writing and checking communication protocols following the *Affine Multiparty Session Types* theory introduced in our ECOOP'22 paper. `MultiCrusty` can work, and should be used, with `Scribble` [2] and `kMC` [1]: with the former tool, users can write *correct global protocols* and project them onto *local* Rust types defined within `MultiCrusty`, this approach is qualified as *top-down*; while the latter tool allows to check *local* Rust types written by users, this approach is qualified as *bottom-up*. Our artifact contains those three tools, their respective source files, as well as the different examples and benchmarks introduced in our paper, all together within a Docker image.

2012 ACM Subject Classification Software and its engineering → Software usability; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi

Keywords and phrases Rust language, affine multiparty session types, failures, cancellation

Digital Object Identifier 10.4230/DARTS.8.2.9

Funding The work is supported by EPSRC 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 NCSS/EPSRC VeTSS.

Acknowledgements We thank the ECOOP reviewers for their insightful comments and suggestions.

Related Article Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida, "Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types", in 36th European Conference on Object-Oriented Programming (ECOOP 2022), LIPIcs, Vol. 222, pp. 4:1–4:29, 2022.
`https://doi.org/10.4230/LIPIcs.ECOOP.2022.4`

Related Conference 36th European Conference on Object-Oriented Programming (ECOOP 2022), June 6–10, 2022, Berlin, Germany

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2022 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

The purpose of this document is to describe in detail the steps required to assess the artifact associated with our paper. We claim our artifact to be *functional*, *reusable* and *available* as followed:

1. **Functionality**: `MultiCrusty` can be used for safe communication programming in Rust. In particular, you should be able to verify three claims from the paper:
   - use `MultiCrusty` to write and verify affine protocols using MPST and Scribble as explained in Section 2 in the paper, i.e top-down approach. **Check the claim by**: following Part II: Step 1.1 (§ Instructions);

- use `MultiCrusty` to write and verify affine protocols using MPST and kMC, i.e bottom-up approach, as explained in Section 2 in the paper. **Check the claim by**: following Part II: Step 1.2 (§ Instructions);
- observe detected errors due to incompatible types, as explained in Section 2 (line 221-225) in the paper. **Check the claim by**: following Part II: Step 1.3 (§ Instructions).

2. **Functionality**: Reproduce the benchmarks in Section 5 (i.e., Table 2 and Figure 9):
   1 claim on expressiveness (Section 5.2 in the paper): examples in Table 2 can be expressed using `MultiCrusty`;
   Check the claim by: Table 2 can be reproduced following the instructions in Part II: Step 2 (§ Instructions);
   2 claim on compile-time performance (line 886-892):
   - the more participants there are, the higher is the compilation time for MPST;

   3 claim on run-time performance (line 880-885):
   - `MultiCrusty` is faster than the BC implementation when there is a large number of interactions and participants (full-mesh protocol)
   - the worst-case scenario for `MultiCrusty` is protocols with many participants but no causalities between them which results in a slowdown when compared with BC. (ring protocol);
   - AMPST has a negligible overhead in comparison to MPST;

   Check claims 2.1 and 2.2 by: Figure 9 can be reproduced following the instructions in Part II: Step 3 (§ Instructions).
3. **Reusability**: The `MultiCrusty` tool can be used to verify your own communication protocols and programs, follow the instructions in Part III (§ Instructions).
4. **Availability**: We agree our artifact to be published under a Creative Commons license on DARTS.

Note on performance. The benchmark data in the paper was generated using a 32-cores AMD OpteronTM Processor 6282 SE machine (the tool makes heavy use of multicore, when available) with a quota of more than 100.000 files and 100 GB of HDD. In particular, measurements in the paper are taken using AMD Opteron Processor 6282 SE @ 1.30 GHz x 32, 128 GiB memory, 100 GB of HDD, OS: ubuntu 20.04 LTS (64-bit), Rustup: 1.24.3, Rust cargo compiler: 1.56.0 Depending on your test machine, the absolute values of the measurements produced in Part II: Step 2 and Step 3 will differ from the paper. Nevertheless, the claims stated in the paper should be preserved.

2 Content

The artifact is submitted as a Docker image. It contains:

- the directory `mpst_rust_github` – a directory containing the source code of the `MultiCrusty` tool;
  - `mpst_rust_github/examples` – contains many examples implemented using `MultiCrusty`; including all examples reported in Figure 9 and Table 2 in the paper;
  - `mpst_rust_github/scripts` – the scripts for reproducing the results;
  - `mpst_rust_github/benches` – the examples for Figure 9.
- the directory `scribble-java` that contains the Scribble source code for generating Rust types from Scribble protocols;
- the directory `kmc` that contains the external kMC tool used to verify that `MultiCrusty` types written in Rust are compatible.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://www.dropbox.com/s/qvl7wijo9n7tcgn/artifact.tar.gz?dl=0`.

4 Tested platforms

This artifact has been tested on Windows, Ubuntu and Mac machines, with at least 16 GB of RAM and 50 GB of disk space. The library itself is lightweight but the examples and benchmarks pose that requirement. The users must also enable localhost access (note that it should be enabled by default unless you disabled it beforehand).

5 License

The artifact is available under a Creative Commons license on DARTS.

6 MD5 sum of the artifact

6c3c0a2f32cee584d068af7590a820f6

7 Size of the artifact

2.1GiB

A Instructions

Hereafter are all the instructions to use and test the artifact.

A.1 Getting started

For the ECOOP'22 artifact evaluation, please use the docker image provided:

0. Install Docker and open the terminal and configure docker setting. **Important**: By Default docker is limited to use only 2GB-4GB RAM, open docker settings and increase the RAM usage to 16GB. See instructions for MacOS and Windows;
1. download the artifact file (assume the filename is `artifact.tar.gz`);
2. unzip the artifact file:

```
gunzip artifact.tar.gz
```

3. you should see the tar file `artifact.tar` after the previous operation;
4. load the docker image:

```
docker load < artifact.tar
```

5. you should see at the end of the output after previous operation:

```
Loaded image: MultiCrusty:latest
```

6. run the docker container:

```
docker run -it --rm MultiCrusty:latest
```

Note: You may need to run the above command with `sudo`.

1. The Docker image comes with an installation of `vim` and `neovim` for editing. If you wish to install additional software for editing or other purposes, you may obtain sudo access with the password `MultiCrusty`;
2. thereafter, we assume that you are in the `mpst_rust_github` directory of the docker file.

A.2 Part I: Quick Start

1. run the tests to make sure `MultiCrusty` is installed and configured correctly:

```
cargo test --tests --all-features --workspace # Test all tests
```

The above command may take up to 15 min.

2. run the examples from Table 2:

```
cargo test --examples --all-features --workspace # Test all examples
```

The above command may take up to 15 min.

3. run the benchmarks from Figure 9:

```
cargo test --benches --all-features --workspace # Test all benchmarks
```

The above command may take up to 15 min. If your command results in an error (error: could not compile `mpstthree`; signal: 9, SIGKILL: kill), this indicated that you do not have a sufficient amount of RAM. Make sure that your docker is configured correctly, i.e open docker settings and increase the RAM usage to 16GB. See instructions for MacOS and Windows.

Note: The commands from steps 1-3 can be run altogether with:

```
cargo test --all-targets --all-features --workspace
# Test everything in the library
```

A.3 Part II: Step by Step instructions

A.3.1 STEP 1: Run the main example (VideoStream) of the paper (Section 2)

In order to run our main example, follow the different steps:

1. check and run the running example from the paper using the top-down approach:
   - execute the following command:

```
./scripts/top_down.sh
```

2. check and run the running example from the paper using the bottom-up approach:
   - execute the following command:

```
./scripts/bottom_up.sh
```

3. edit the program and observe the reported errors.

Next, we highlight how concurrency errors are ruled out by `MultiCrusty` (i.e., the ultimate practical purpose of `MultiCrusty`). After each modification, compile the program with `cargo run --example=video_stream_generated --features="baking_checking"` and observe the reported error.

- open the file video_stream_generated.rs in the `examples/` folder, containing the *VideoStream* program, with your favourite text editor.

Suggested modifications:

- swap lines 104 and 105 (this can lead to a deadlock);
- use another communication primitive, replace `let (video, s) = s.recv()?;` on line 106 with `let s = s.send(0)?;` – compilation errors because type mismatch;
- keep the changes from the previous modification and in addition modify the types at line 17, corresponding to line 106, from `Recv` to `Send` – mismatch because of duality.

A.3.2 STEP 2: Running the examples from Table 2

The purpose of these examples is to demonstrate how the tool works on existing examples from the literature.

The examples in this table are located in the folder `examples/`.

The data for these benchmarks can be re-generated using the following script:

```
./scripts/examples_literature.sh
# Will take up to one hour, progress is displayed in the terminal
```

Each command is run 10 times on each example and the columns display the means in *ms*.

Results are outputted in the file `results/benchmarks_main_from_literature_0.csv` where we give in brackets the corresponding names from Table 2 in the paper:

- column 1: file name (Example/Endpoint);
- column 2: **check** time in **microseconds**, the result of `cargo check` (Check);
- column 3: **build** time in **microseconds**, the result of `cargo build` (Comp.);
- column 4: **build –release** time in **microseconds**, the result of `cargo build --release` (Rel.);
- column 5: **run** time in **nanoseconds**, the result of running `cargo bench` (Exec time).

A.3.3 STEP 3: Running benchmarks from Figure 9 (ping-pong, mesh and ring protocols)

The purpose of this set of benchmarks is to demonstrate the scalability of the tool on large examples.

A.3.3.1 Option 1: Running a small benchmark set

You can run a small set of the benchmarks since the full benchmark set can take about 24 hours. We have prepared a lighter version that should complete in about three hours. The difference is that `ping_pong` protocols are run up to 50 loops (and not 500), and `mesh` and `ring` protocols are up to *five* participants (and not *ten*). Each benchmark has a significance of 0.1 and a sample size of 100 in this configuration: each protocol is run 100 times.

These modifications are enough to start observing the performance trends (refer to claims about functionality at the beginning of this document).

To run the lighter benchmark:

```
./scripts/lightweight_library.sh # Set up
```

then by running the command line:

```
./scripts/ping_pong_mesh_ring_light.sh # This will take up to 3 hours
```

Results. After running the above scripts, the graphs are saved in the `results/` folder in the file `graphs_0.pdf`, alongside the raw data for the graphs (.csv files).

To open the `graphs_0.pdf` file, copy the `results/` folder to a local directory on your machine.

For detailed instructions on how to copy a docker folder to a local folder check here.

In short, open a terminal, type `docker ps` to check the name of the running docker container for MultiCrusty:latest. The command should return the id of the container, let assume it is c4a9485b3222. Then given that "Documents/Docker" is a local directory in your system, execute the command:

```
docker cp c4a9485b3222:"home/MultiCrusty/mpst_rust_github/results"\
    "Documents/Docker"
```

The above will copy the results folder from the docker container to your directory Documents/Docker. Open the file graphs_0.pdf, it will contain 5 graphs that correspond to the graphs displayed in Figure 9.

Details on the content of the raw .csv files data (optional reading).

The `ping_pong_mesh_ring_light.sh` and `ping_pong_mesh_ring_full.sh` scripts generate 3 files: `ping_ping_0.csv`, `mesh_0.csv` and `ring_0.csv` in the folder `results/`.

The structure of the `ping_ping_0.csv` file is as follows:

1. column 1: the type of implementation (`AMPST`, `MPST`, `binary` or `crossbeam`);
2. column 2: number of loops;
3. column 3: average running time (in nanosecond);
4. column 4: average compilation time (in microseconds).

The structure of the `mesh_0.csv` and `ring_0.csv` files are as follows:

1. column 1: the type of implementation (`AMPST`, `MPST`, `binary` or `crossbeam`);
2. column 2: number of participants;
3. column 3: average running time (in nanosecond);
4. column 4: average compilation time (in microseconds).

Be aware that the scripts add additional `*.csv` files on top of the existing ones.

A.3.3.2 Option 2: Running the entire benchmark set (at least 24 hours)

To run the same set of benchmarks as in the paper, i.e ping-pong for up to 500 iterations and ring and mesh for 10 participants, execute the following commands:

```
./scripts/full_library.sh # set up
```

Then you can run the script:

```
./scripts/ping_pong_mesh_ring_full.sh # This will take more than 24 hours
```

Each benchmark has a significance of 0.1 and a sample size of 10000 in this configuration.

Note: we have executed this script on a high-performance computing server, and running the whole script took over 24 hours. Progress is shown while running each benchmark.

You can also run one of the following scripts to retrieve results for only one kind of protocol:

```
./scripts/ping_pong.sh # For ping-pong protocols
##
./scripts/mesh_full.sh # For mesh protocols with full_library.sh
./scripts/ring_full.sh # For ring protocols with full_library.sh
##
./scripts/mesh_light.sh # For mesh protocols with lightweight_library.sh
./scripts/ring_light.sh # For ring protocols with lightweight_library.sh
```

A.4 Part III: A walkthrough tutorial on checking your own protocols with `MultiCrusty`

You can write your own examples using (1) generated types from `Scribble` (top-down approach) or (2) your own types written with `MultiCrusty` and then check them using the kMC tool (bottom-up approach).

A.4.1 3.1 Top-down: Generating Types from Scribble

In the `top-down` approach, protocols written in the protocol description language `Scribble` are used for generating `MultiCrusty` types.

You can use our implementation of a simple recursive protocol that forwards (adds) a number between three participants. The protocol is provided in the `Scribble` repository as a start. The protocol is located in `scribble-java/scribble-demos/scrib/fib/src/fib/Fib.scr`

Follow the steps to implement a simple *adder* example with Scribble and `MultiCrusty`:

1] Generate Rust Types from Scribble:

```
./scripts/top_down_adder.sh
```

In the above example, we move into the `scribble-java` folder and run the `Scribble` API for `Rust` on the `Adder` protocol written with `Scribble`. This command outputs the file `adder_generated.rs` at the root of the `scribble-java` directory. Then it moves the file `adder_generated.rs` from the `scribble-java` folder to the `examples` subfolder of the `mpst_rust_github` folder containing `MultiCrusty` and auto-format the file with `cargo fmt`.

Now, you can open the `examples/adder_generated.rs` file using your preferred editor program before testing the protocol directly with `MultiCrusty`.

→ From this point, we assume that you will remain in the `MultiCrusty` repository (the `mpst_rust_github` folder).

2] Compile the Rust types:

```
cargo run --example=adder_generated --features="baking"
```

This command contains four parts:

1. `cargo` which calls the `Rust` compiler;
2. `run` for compiling and running one or more `Rust` files;
3. `--example=adder_generated` for running the specific; *example* `adder_generated`;
4. `--features="baking"` for compiling only specific parts of `MultiCrusty` used for the example.

You will have an error and several warnings when running the previous command. This is because the `Scribble` API only generates `Rust` types and the `Rust` compiler needs at least a `main` function.

Hereafter, we provide the code for the processes that implement the generated types.

3] Implement the endpoint programs for role `A`, `B` and `C`:

```rust
//////////////////////////

fn endpoint_a(s: EndpointA48) -> Result<(), Box<dyn Error>> {
    let (_, s) = s.recv()?;
    offer_mpst!(s, {
        BranchesOAtoC::Add(s) => {
            recurs_a(s)
        },
        BranchesOAtoC::Bye(s) => {
            let (_,s) = s.recv()?;
            s.close()
        },
    })
}

fn recurs_a(s: EndpointA23) -> Result<(), Box<dyn Error>> {
    let (_, s) = s.recv()?;
    offer_mpst!(s, {
        BranchesOAtoC::Add(s) => {
            recurs_a(s)
        },
        BranchesOAtoC::Bye(s) => {
            let (_,s) = s.recv()?;
            s.close()
        },
    })
}

//////////////////////////

fn endpoint_b(s: EndpointB50) -> Result<(), Box<dyn Error>> {
    offer_mpst!(s, {
        BranchesOBtoC::Add(s) => {
            let (_,s) = s.recv()?;
            let s = s.send(0)?;
            endpoint_b(s)
        },
        BranchesOBtoC::Bye(s) => {
            let (_,s) = s.recv()?;
            let s = s.send(())?;
            s.close()
        },
    })
}

//////////////////////////

fn endpoint_c(s: EndpointC13) -> Result<(), Box<dyn Error>> {
```

```
    let s = s.send(0)?;
    recurs_c(s, 5)
}

fn recurs_c(s: EndpointC10, loops: i32) -> Result<(), Box<dyn Error>> {
    if loops <= 0 {
        let s: EndpointC7 = choose_mpst_c_to_all!(
            s, Branches0AtoC::Add, Branches0BtoC::Add);
        let s = s.send(0)?;

        recurs_c(s, loops - 1)
    } else {
        let s: EndpointC9 = choose_mpst_c_to_all!(
            s, Branches0AtoC::Bye, Branches0BtoC::Bye);
        let s = s.send(())?;

        s.close()
    }
}

//////////////////////////

fn main() {
    let (thread_a, thread_b, thread_c) = fork_mpst(endpoint_a, endpoint_b, endpoint_c);

    assert!(thread_a.join().is_ok());
    assert!(thread_b.join().is_ok());
    assert!(thread_c.join().is_ok());
}
```

There are four different parts: the first three ones are for representing the different roles, `A`, `B` and `C`, involved in the protocol and the last one (the main function) runs all processes together.

In the first three parts, we are using the primitives described in Table 1 of the paper:

1. `send(p)` for sending a payload `p`;
2. `recv()` for receiving a payload;
3. `offer_mpst!` for receiving a choice;
4. `choose_mpst_c_to_all!` for sending a choice.

The main function uses `fork_mpst` to fork the different threads.

All those primitives are generated using the macro `bundle_impl_with_enum_and_cancel!`.

Now, if you run again the file, it should run correctly:

```
cargo run --example=adder_generated --features="baking"
```

A.4.2 3.2 Bottom-up: Write the types in Rust and check them with the kmc tool

Adder example with kMC. We show how to use the bottom-up approach. The first step in the bottom-up approach to write the Rust types for the meshed channels. We will use the Adder

example from above, since we already have the types, and we will only demonstrate here how to check them using the external kMC tool.

`MultiCrusty` uses the macro `checker_concat!` on the types to rewrite Rust types to communicating finite state machines (CFSM) that the `kMC` checks.

This macro also returns the CFSM (visual) representation for each type using the **dot** format.

Now, that you have a better idea of the interactions between those two tools, we will check the types in the `adder_generated` example are correct using our macro `checker_concat!`.

For this purpose, append the following lines to the `adder_generated.rs` file:

```
//////////////////////

fn checking() {
    let (graphs, kmc) = mpstthree::checker_concat!(
        "adder_checking",
        EndpointA48,
        EndpointC13,
        EndpointB50
        =>
        [
            EndpointC7,
            BranchesOAtoC, Add,
            BranchesOBtoC, Add,
        ],
        [
            EndpointC9,
            BranchesOAtoC, Bye,
            BranchesOBtoC, Bye,
        ]
    )
    .unwrap();

    println!("graph A: {:?}", petgraph::dot::Dot::new(&graphs["RoleA"]));
    println!("\n//////////////////////////\n");
    println!("graph B: {:?}", petgraph::dot::Dot::new(&graphs["RoleB"]));
    println!("\n//////////////////////////\n");
    println!("graph C: {:?}", petgraph::dot::Dot::new(&graphs["RoleC"]));
    println!("\n//////////////////////////\n");
    println!("min kMC: {:?}", kmc);
}
```

and update the `main()` function by including `checking();` in it:

```
fn main() {
    checking();

    let (thread_a, thread_b, thread_c) =
         fork_mpst(endpoint_a, endpoint_b, endpoint_c);

    assert!(thread_a.join().is_ok());
```

```
    assert!(thread_b.join().is_ok());
    assert!(thread_c.join().is_ok());
}
```

Now, if you run again the file, it should run correctly:

```
cargo run --example=adder_generated --features="baking_checking"
```

Notice the different features used for compiling the example: **baking__checking** instead of **baking**.

If you are unsure about either of the above steps, the `Rust` code is available in the `adder.rs` file located in the `examples/` folder.

_Optional: If you want more practice writing types and programs using `MultiCrusty`, and kMC, check the additional examples section at the end of the document: A simple example with `MultiCrusty` and kMC in the Additional Information section.

A.5 ADDITIONAL INFORMATION

Benchmark setup in the paper. All set-up and benchmarks were performed on the following machine:

- AMD OpteronTM Processor 6282 SE @ 1.30 GHz x 32, 128 GiB memory, 100 GB of HDD, OS: ubuntu 20.04 LTS (64-bit), Rustup: 1.24.3, Rust cargo compiler: 1.56.0.

The original benchmarks were generated using:

- compile and run: `cargo bench --all-targets --all-features --workspace`

Generating documentation for `MultiCrusty`. The documentation of `MultiCrusty` can be generated with the command `cargo doc --all-features`.

The generated documentation will be accessible in the file `target/doc/mpstthree/index.html`.

The source code is included in the root directory.

Rust commands on build, test, compile. Here is a general description of all commands you can run to check, build and test.

```
cd mpst_rust_github # Move to MultiCrusty's repository
cargo check --all-features --lib --workspace # Check only this package's library
cargo check --all-features --bins --workspace # Check all binaries
cargo check --all-features --examples --workspace # Check all examples
cargo check --all-features --tests --workspace # Check all tests
cargo check --all-features --benches --workspace # Check all benchmarks
cargo build --all-features --lib --workspace # Build only this package's library
cargo build --all-features --bins --workspace # Build all binaries
cargo build --all-features --examples --workspace # Build all examples
cargo build --all-features --tests --workspace # Build all tests
cargo build --all-features --benches --workspace # Build all benchmarks
cargo test --all-features --lib --workspace # Test only this package's library
cargo test --all-features --bins --workspace # Test all binaries
cargo test --all-features --examples --workspace # Test all examples
cargo test --all-features --tests --workspace # Test all tests
cargo test --all-features --benches --workspace # Test all benchmarks
```

Scribble commands. Assuming you know how to write `Scribble` protocols, put your own in the folder `../scribble-java/scribble-demos/scrib/fib/` and use:

```
# Move to the correct repository
cd scribble-java/
# Run Scribble on the global protocol to project it onto local Rust types
./scribble-dist/target/scribblec.sh -ip scribble-demos/scrib/fib/src
    -d scribble-demos/scrib/fib/src
         scribble-demos/scrib/fib/src/fib/[input file without extension].scr
            -rustapi [name of the protocol] [output file without extension]
# Move back to the previous repository
cd ..
# Move the generated file to the example folder
mv scribble-java/[input file without extension].rs
      mpst_rust_github/examples/[output file without extension].rs
# Move to the mpst_rust_github/ repository
cd  mpst_rust_github/
```

A simple example with `MultiCrusty` and kMC.
Need help? This example is implemented in `examples/basic.rs`, hence you can use the file as a reference implementation.

1] Import the necessary macros from the `MultiCrusty` library:

```
// The basic types
use mpstthree::binary::struct_trait::{end::End, recv::Recv, send::Send};
// The macro for generating the roles and the MeshedChannels
use mpstthree::baker;
// Optional: used only for protocols with choice/offer
use mpstthree::role::broadcast::RoleBroadcast;
// The final type for the stacks and the names of the roles
use mpstthree::role::end::RoleEnd;
// Used for checking the protocol
use mpstthree::checker_concat;
// Used for functions output
use std::error::Error;
```

2] Create the **roles** and the **MeshedChannels** data structure:

```
// generates meshed channels for 3 roles
bundle_impl_with_enum_and_cancel!( MeshedChannels, A, B);
```

The new generated types will be `MeshedChannels` and `RoleX` where `X` is the provided name in the macro inputs.

3] Write the **MeshedChannels** types:

A good practice is to write the simplest types first, and concatenate them into `MeshedChannels`. That is why we first write down the types used for representing the roles:

```
// Payload types
struct Request;
struct Response;
struct Stop;
```

```
// Names
type NameA = RoleA<RoleEnd>;
type NameB = RoleB<RoleEnd>;
```

Then we write each binary type:

```
// Binary types for A
// Recv a Request then Send a choice
type StartA0 = Recv<Request, Send<Branching0fromAtoB, End>>;
// Stack for recv then sending a choice
type OrderingA0 = RoleB<RoleBroadcast>;

// Send a choice
type LoopA0 = Send<Branching0fromAtoB, End>;
// Stack for sending a choice
type OrderingLoopA0 = RoleBroadcast;

// Recv Response then send a choice
type MoreA1 = Recv<Response, Send<Branching0fromAtoB, End>>;
// Stack for the previous binary type
type OrderingMoreA1 = RoleB<RoleBroadcast>;

// Recv Stop
type DoneA1 = Recv<Stop, End>;
// Stack for the previous binary type
type OrderingDoneA1 = RoleB<RoleEnd>;

// Binary types for B
// Send a Request then Recv a choice
type StartB0 = Send<Request, Recv<Branching0fromAtoB, End>>;
// Stack for send then receiving a choice from A
type OrderingB0 = RoleA<RoleA<RoleEnd>>;

// Recv a choice
type LoopB0 = Recv<Branching0fromAtoB, End>;
// Stack for recv a choice
type OrderingLoopB0 = RoleA<RoleEnd>;

// Recv Request then Send Response then receive a choice
type MoreB1 = Send<Response, Recv<Branching0fromAtoB, End>>;
// Stack for the previous binary type
type OrderingMoreB1 = RoleA<RoleA<RoleEnd>>;

// Send Stop
type DoneB1 = Send<Stop, End>;
// Stack for the previous binary type
type OrderingDoneB1 = RoleA<RoleEnd>;

// Sum type containing the different paths of the choice
```

```
enum BranchingOfromAtoB {
    More(MeshedChannels<MoreB1, OrderingMoreB1, NameB>),
    Done(MeshedChannels<DoneB1, OrderingDoneB1, NameB>),
}
```

This protocol is recursive as you may have noticed with `MoreB1` both inside the `enum` type `BranchingOfromAtoB` and containing `Recv<BranchingOfromAtoB, End>`. The two paths are `More` and `Done`.

We are now going to concatenate the previous types into `MeshedChannels`:

```
// Creating the endpoints
// A
type EndpointAMore = MeshedChannels<MoreA1, OrderingMoreA1, NameA>;
type EndpointADone = MeshedChannels<DoneA1, OrderingDoneA1, NameA>;
type EndpointALoop = MeshedChannels<LoopA0, OrderingLoopA0, NameA>;
type EndpointA = MeshedChannels<StartA0, OrderingA0, NameA>;

// B
type EndpointBLoop = MeshedChannels<LoopB0, OrderingLoopB0, NameB>;
type EndpointB = MeshedChannels<StartB0, OrderingB0, NameB>;
```

4] Check that the types are correct:

We can check that the written types are compatible using the `checker_concat!` macro which translates the types to Communicating Finite State machines (CFSM) and uses the kMC tool to check for compatibility. Note that, in practice, since this is a binary protocol, we do not need to invoke the kMC tool, since the duality between the types is enough to guarantee correctness.

```
fn main() {
    let (_, kmc) = checker_concat!(
        "basic",
        EndpointA,
        EndpointB
        =>
        [
            EndpointAMore,
            BranchingOfromAtoB, More,
        ],
        [
            EndpointADone,
            BranchingOfromAtoB, Done,
        ]
    )
    .unwrap();

    println!("min kMC: {:?}", kmc);

    // let (thread_a, thread_b) = fork_mpst(endpoint_a, endpoint_b);

    // assert!(thread_a.join().is_ok());
    // assert!(thread_b.join().is_ok());
}
```

Run the checker_concat! macro to check if the types are correct:

```
cargo run --example=my_basic --features="baking_checking"
```

After running the command above, the terminal should display the output from the kMC tool, which is the minimal **k** for this protocol. It is **1** for the protocol, as expected.

5] Implement the endpoint processes for `A`, `B` by adding the following code after the **main** function:

```
fn endpoint_a(s: EndpointA) -> Result<(), Box<dyn Error>> {
    let (_, s) = s.recv()?;
        recurs_a(s, 5)
    }

fn recurs_a(s: EndpointALoop, loops: i32) -> Result<(), Box<dyn Error>> {
    if loops > 0 {
        let s: EndpointAMore = choose_mpst_a_to_all!(s, BranchingOfromAtoB::More);

        let (_, s) = s.recv()?;
        recurs_a(s, loops - 1)
    } else {
        let s: EndpointADone = choose_mpst_a_to_all!(
            s, BranchingOfromAtoB::Done);

        let (_, s) = s.recv()?;
        s.close()
    }
}

fn endpoint_b(s: EndpointB) -> Result<(), Box<dyn Error>> {
    let s = s.send(Request {})?;
        recurs_b(s)
    }

fn recurs_b(s: EndpointBLoop) -> Result<(), Box<dyn Error>> {
    offer_mpst!(s, {
        BranchingOfromAtoB::More(s) => {
            let s = s.send(Response {})?;
            recurs_b(s)
        },
        BranchingOfromAtoB::Done(s) => {
            let s = s.send(Stop {})?;
            s.close()
        },
    })
}
```

Finally, uncomment the last three lines in the **main** function by removing the `//` at the beginning of each line.

6] Run the example again:

```
cargo run --example=my_basic --features="baking_checking"
```

References

1 Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117, Cham, 2019. Springer. doi:10.1007/978-3-030-25540-4_6.

2 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, pages 22–41, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-05119-2_3.