Session Types for the Transport Layer: Towards an Implementation of TCP*

Samuel Cavoj
samuel@cavoj.net
University of Glasgow

Ivan Nikitin
ivan@niktivan.org
University of Glasgow

Colin Perkins
csp@csperkins.org
University of Glasgow

Ornela Dardha
ornela.dardha@glasgow.ac.uk
University of Glasgow

Session types are a typing discipline used to formally describe communication-driven applications with the aim of fewer errors and easier debugging later into the life cycle of the software. Protocols at the transport layer such as TCP, UDP, and QUIC underpin most of the communication on the modern Internet and affect billions of end-users. The transport layer has different requirements and constraints compared to the application layer resulting in different requirements for verification. Despite this, to our best knowledge, no work shows the application of session types at the transport layer. In this work, we discuss how multiparty session types (MPST) can be applied to implement the TCP protocol. We develop an MPST-based implementation of a subset of a TCP server in Rust and test its interoperability against the Linux TCP stack. Our results highlight the differences in assumptions between session type theory and the way transport layer protocols are usually implemented. This work is the first step towards bringing session types into the transport layer.

1 Introduction

Session types [11] are a typing discipline for communication protocols. They can describe the sequence of messages exchanged between participants over a communication channel and can be used to verify that the protocol is implemented correctly or has certain desirable properties. Further, session types can be realised within programming languages and used to type-check the implementation of a protocol against a session type definition, with type errors indicating inconsistencies between implementation and the session type. Session types have been an active area of research since the beginning of the 1990s [11] and have been implemented in a number of programming languages including C [26], Java [13] and Rust [14, 15] and other programming languages [9, 16, 25, 27, 29].

Network protocols that are part of the Internet Protocol suite (TCP/IP) are the foundation of the Internet. They are responsible for interoperability between different devices, operating systems, and applications. To ensure that different implementations of the same protocol are compatible, they must adhere to a technical specification which, in the case of Internet protocols, is defined in a series of documents, known as RFCs [8], developed by the Internet Engineering Task Force (IETF). Specifically, the latest version of the TCP protocol specification is defined in RFC 9293 [7].

The IETF follows a consensus-based process when developing standards [4, 30], with protocol specifications being developed in working group meetings and on mailing lists over a multi-year period. The resulting RFCs are written primarily in English prose, allowing the documents to be used in the

*Supported in part by the UK EPSRC grants EP/X027309/1 and EP/S036075/1.

D. Costa, R. Hu (Eds.): Programming Language Approaches to
Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. 22–36, doi:10.4204/EPTCS.401.3

consensus-building process, but the natural language can be ambiguous and unclear and this can lead to inconsistent and non-conforming implementations. [22, 23, 28]. In this sense, ensuring the correctness of Internet protocols is vital. Developing formalised models of the protocols described in RFCs is one way to achieve this. Session types are one such modelling technique that has not previously been explored for transport-layer protocols, such as TCP.

In this paper, we implement a core subset of the TCP protocol in the Rust programming language and use session types to describe the network operations. Session types are encoded into native Rust types and the type checker is used to verify that the implementation follows the session type specification. In this way, the Rust compiler verifies that the implementation of the protocol is correct in terms of the types of messages exchanged and the order in which they are exchanged, i.e., that it follows the declared session type, for a session type model describing a synchronous subset of TCP. Additionally, session types are used to describe the application interface, so we can verify that the application uses the TCP implementation correctly.

Our contributions are as follows:

1. **Session Types Libraries.** We develop[1] the libraries required for encoding the session type model into native Rust types in an ergonomic fashion (§4.1).
2. **Implementation.** We implement a subset of the TCP protocol [7], including key aspects of both the user/TCP interface and the TCP/lower-level interface, in Rust while adhering to the session type model. This is done in a way such that the Rust compiler can detect deviation from the session type (§4.4).
3. **Testing.** We test our implementation against a real TCP stack (§5).

The remainder of this paper is structured as follows. Section 2 briefly reviews the multiparty session type model we use. Section 3 outlines key properties of TCP and its state machine. Section 4 describes our session typed implementation of TCP in Rust. Section 5 evaluates the correctness of our implementation. Finally, Section 6 reviews related work and concludes.

2 Session types

Session types [11] describe communication among participants in a distributed system in terms of the types and order of messages that are exchanged. A single session type describes the sequence of messages sent or received from the perspective of one of the participants. The theory of session types was later extended to multiparty session types (MPST) which can describe protocols between any number of participants [12].

In this paper, the bottom-up multiparty session type approach [31] is used to describe TCP. An example of a simple ping-pong protocol using this approach is demonstrated in Equation 1. When type-checking any type using the bottom-up approach, we must additionally choose a *safety invariant*. Safety invariants are parameters associated with the properties a protocol may demonstrate during runtime, such as deadlock-freedom and liveness. Each safety invariant is accompanied by specific typing rules (not presented here) that guarantee the maintenance of the corresponding invariant. If the protocol successfully type-checks with the instantiation of the safety invariant, it will manifest the property represented by the invariant during its runtime.

[1] Our session type library and TCP implementation is available at `https://github.com/sammko/tcpst2`

$$\begin{aligned}\Gamma_1 = \quad & \mathtt{s}[\mathtt{a}] : \mathtt{b} \oplus l_1(\mathtt{ping}) \,.\, \mathtt{b} \,\&\, l_3(\mathtt{pong}) \,.\, \mathtt{end}, \\ & \mathtt{s}[\mathtt{b}] : \mathtt{a} \,\&\, l_2(\mathtt{ping}) \,.\, \mathtt{a} \oplus l_1(\mathtt{pong}) \,.\, \mathtt{end}\end{aligned} \qquad (1)$$

The implications of this approach are that global types and the concept of duality are not used. Instead of duality, the compatibility invariant is used to check that actions are dual between the given types. However, a protocol can still be described using session types even if safety does not hold.

3 Transmission Control Protocol (TCP)

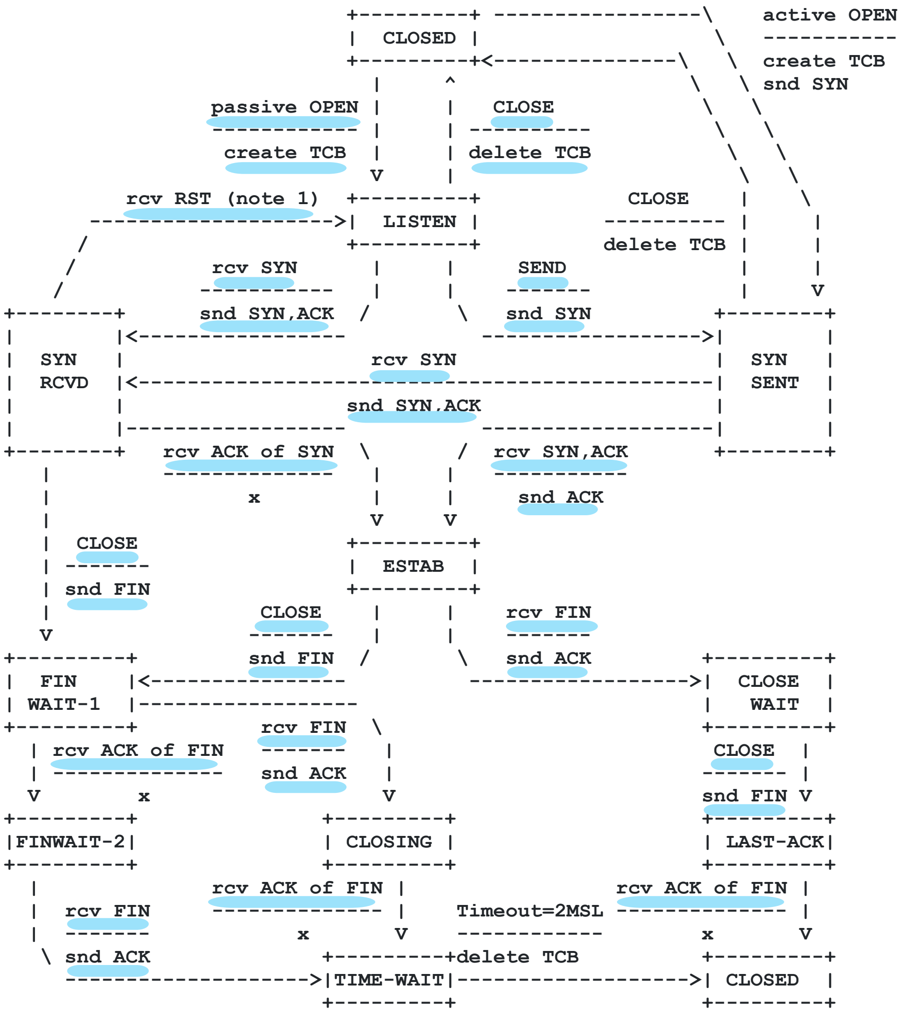
The TCP transport is layered on top of the datagram service provided by the Internet Protocol (IP). The IP layer provides an unreliable, best-effort, datagram service, where packets may be lost, duplicated, delayed, or re-ordered in transit. TCP segments, sent within IP packets, contain sequence numbers and acknowledgements such that, upon detection of a lost packet, either triggered by a timer expiration or receipt of a triple-duplicate acknowledgement, the sender can re-transmit the lost segment.

TCP is usually used in a client-server manner, but also supports a rarely used simultaneous open mode with peer-to-peer connections. In the context of this paper, we assume client-server usage, with one side being a passive server listening for incoming connections, while the other is an active client initiating the connection. We describe the operation of the TCP state machine below and provide a diagram of the TCP state transitions in Figure 1.

The establishment of a reliable connection between two network devices is facilitated by the TCP three-way handshake. It commences with the initiation of a connection with the client sending a TCP segment with the SYN (synchronise) bit set in the header and containing the client's initial sequence number. The server responds with a segment with the SYN and ACK bits set, acknowledging the client's initial sequence number and providing the initial sequence number the server will use. Finally, the client confirms the establishment of the connection by sending a segment with the ACK (acknowledge) bit set. This sequence ensures both sides agree on their initial sequence numbers and confirm their willingness to communicate.

TCP uses a sliding window algorithm to manage data transmission by sending segments with sequence numbers. The window size determines the number of unacknowledged segments in transit. The receiver discards unacceptable segments falling outside the expected sequence range, leading to retransmission by the sender. Acknowledgements are sent upon receiving new data, indicating the next expected contiguous sequence number. TCP handles packet loss or reordering at the IP layer by detecting duplicate acknowledgements; a triple-duplicate acknowledgement triggers retransmission. Additionally, TCP utilises a retransmission timeout (RTO) mechanism, dynamically adjusted based on network conditions. TCP buffers play a crucial role on both the sender and receiver sides, with the send buffer holding outgoing segments awaiting acknowledgement and the receive buffer storing incoming segments yet to be delivered to the application.

The TCP closing handshake, another three-way handshake involving packets with the FIN (finish) and ACK (acknowledge) bits signifies the end of a connection. The initial party sends a FIN packet, followed by an acknowledgement from the other party, culminating in a reciprocal FIN-ACK exchange. The final step includes an acknowledgement from the original sender, leading to the TIME-WAIT state. This state ensures a reliable closure, allowing the handling of delayed or duplicate IP packets before concluding the connection.



```
                            +---------+ ---------\      active OPEN
                            |  CLOSED |            \    -----------
                            +---------+<---------\   \   create TCB
                              |     ^              \   \  snd SYN
                 passive OPEN |     |   CLOSE        \   \
                 ------------ |     | ----------       \   \
                  create TCB  |     | delete TCB         \   \
                              V     |                      \   \
          rcv RST (note 1)  +---------+            CLOSE    |    \
       -------------------->|  LISTEN |          ---------- |     |
      /                     +---------+          delete TCB |     |
     /           rcv SYN      |     |     SEND              |     |
    /           -----------   |     |    -------            |     V
+--------+      snd SYN,ACK  /       \   snd SYN          +--------+
|        |<-----------------           ------------------>|        |
|  SYN   |                    rcv SYN                     |  SYN   |
|  RCVD  |<-----------------------------------------------|  SENT  |
|        |                  snd SYN,ACK                   |        |
|        |------------------           -------------------|        |
+--------+   rcv ACK of SYN  \       /  rcv SYN,ACK       +--------+
   |         --------------   |     |   -----------
   |                x         |     |     snd ACK
   |                          V     V
   |  CLOSE                 +---------+
   | -------                |  ESTAB  |
   | snd FIN                +---------+
   |                 CLOSE    |     |    rcv FIN
   V                -------   |     |    -------
+---------+         snd FIN  /       \   snd ACK         +---------+
|  FIN    |<----------------          ------------------>|  CLOSE  |
| WAIT-1  |------------------                            |   WAIT  |
+---------+          rcv FIN  \                          +---------+
  | rcv ACK of FIN   -------   |                          CLOSE  |
  | --------------   snd ACK   |                         ------- |
  V        x                   V                         snd FIN V
+---------+               +---------+                    +---------+
|FINWAIT-2|               | CLOSING |                    | LAST-ACK|
+---------+               +---------+                    +---------+
  |              rcv ACK of FIN |                 rcv ACK of FIN |
  |  rcv FIN     -------------- |    Timeout=2MSL -------------- |
  |  -------            x       V    ------------        x       V
   \ snd ACK              +---------+delete TCB          +---------+
     -------------------->|TIME-WAIT|------------------->| CLOSED  |
                          +---------+                    +---------+
```

Figure 1: The state transition diagram of TCP in RFC9293 [7]. We annotate the diagram with the messages and transitions modelled in our implementation. Note that we do not model timeouts as part of the type system, hence, the TIME-WAIT to CLOSED transition is not implemented using session types. Additionally, we do not implement the active OPEN case of the handshake for simplicity (as this would not demonstrate any new modelling or implementation techniques), this is however possible using our implementation.

4 Implementation

We implement the basic functionality of the TCP server protocol while modelling both the network and the application interface using session types. Note that more information on the implementation can be found in the Appendix. Under the *less is more* formalisation of multiparty session types [31], the roles we are considering the following:

Server User The server application using the TCP protocol.

Server System The TCP implementation.

Client System The TCP implementation on the other end of the network.

The channel between the Server System and the Client System represents the network. The messages exchanged between the Server User and the Server System are a formalisation of the user/TCP (i.e., application programming) interface and do not pass over the network. The system call interfaces, representing the user and the system (in this paper simulated through threads), each have a session type which prescribes their behaviour relative to the other roles. The Client System role has no associated session type in our implementation as it is assumed to be another host on the Internet and not part of our program.

4.1 Defining session types

The basic building blocks of our implementation are the generic structs `OfferOne`, `OfferTwo`, `SelectOne`, and `SelectTwo`. All of these implement the trait `Action` which represents a general session type. The type parameters of the structs encode the role the action is performed with respect to, the types of messages exchanged, and the continuation of the session. In addition, the `End` struct is also an `Action` and represents the end session type.

The `OfferTwo` struct has five type parameters. The first is the peer role, and the next two are the types of messages exchanged in either of the two branches of the offer and the final two parameters are the session types of the continuations of the two branches.

```
pub struct OfferTwo<R, M1, M2, A1, A2>
where R: Role, M1: Message, M2: Message,
    A1: Action, A2: Action,
{
    phantom: PhantomData<(R, M1, M2, A1, A2)>,
}
```

The `OfferTwo` struct, as a way of encoding a session type construct in Rust, has type parameters but contains no data. The `PhantomData`-typed field contained within the struct is a zero-sized marker type that simulates a field of the given type to support the Rust type checker.[2]

The `SelectTwo` struct has the same type parameters and is also a zero-sized type. Finally, the non-branching actions `OfferOne` and `SelectOne` have only three type parameters: the peer role, the message type, and the continuation type, but are otherwise analogous.

To define a session type one can define a type alias for the root action of the session. For example a simple session type for a client-server interaction could be defined as follows:

```
type ServerSt = OfferOne<Client, Request, SelectOne<Client, Response, End>>;
type ClientSt = SelectOne<Server, Request, OfferOne<Server, Response, End>>;
```

This basic syntax, however, quickly becomes unwieldy when defining more complex session types. To address this, we have implemented a macro which converts a more readable syntax into the full definition of the type. Rust's `macro_rules!` mechanism is powerful enough to allow us to define a syntax which attempts to mimic the mathematical notation. The macro is called `St!` and the `ServerSt` type from the above example could be re-written as follows:

```
type ServerSt = St![(Client & Request).(Client + Response).end]
```

[2] `https://doc.rust-lang.org/nomicon/phantom-data.html`

The macro is recursive and supports arbitrary nesting of offers and selections. The full definition can be found in the `st_macros.rs` file of the source code.

4.2 Multi-way `Offer` branching

As Rust does not support variadic generic types, we are not aware of a way to implement a generic `Offer` type which would support a variable number of branches. Hence we implement `OfferOne` and `OfferTwo` as separate constructs with some repetition in the corresponding infrastructure such as the `offer_one`, `offer_two` and similar selection methods. These are described in §4.4.

However, support for more than two branches is required in practice. A simple way to do this is to implement `OfferThree`, `OfferFour`, ..., in the same way, along with the code supporting this. This leads to more code duplication, but does not increase complexity, and the usage is straightforward.

As an alternative, to avoid duplication, we chose a nesting approach where a branching of arity N is transformed into a two-way branching between the first case and an $N-1$ branching of the other cases.[3] This is recursively expanded until it finally results in a tree of two-way forks, where each *left* branch represents a single case from the original N. All *right* branches except the bottom-most one lead to a virtual node which was not present in the original type.

4.3 Recursive session types

Type aliases in Rust cannot be recursive. The reason for this is that a type alias does not create a new type and is merely another name for the same type. For instance, defining a type alias `type A = X<A>` is not allowed because the expansion would be infinite – the name `X<A>` would expand to `X<X<A>>`, etc. However, we somehow need to represent recursive session types.

Fortunately, this is not difficult to circumvent. Whereas type *aliases* cannot be recursive, there is no such restriction for types themselves, as long as the size of the type is finite. As such, types which contain a recursive cycle with no indirection are not allowed as the size of the type is infinite. But inserting indirection into the cycle (such as a reference `&T` or `Box<T>`) resolves this problem since the size of a reference does not depend on the size of the target type `T`.

4.4 Using session types

A channel provides methods to send and receive messages which consume a corresponding session type and return the continuation. The type of the channel is generic over the roles between which it exists and the method signatures ensure that they can be only called with an appropriate session type instance and message. Consider a channel of type `Channel<R1, R2>` which we define as the endpoint belonging to role `R1`, i.e. it can send to or receive from `R2`. Then its `select_one` method could have the following signature:

```
fn select_one<M, A>(&mut self, _o: SelectOne<R2, M, A>, message: M) -> A
where M: Message, A: Action;
```

It is generic over the message type, but it has to match the one prescribed by the provided session typed *token*. The role `R2` is already bound by the channel type. The token is moved into this function, so the owner cannot re-use it. The continuation type from the token is instantiated and returned to the

[3] Naturally, it would be better to split into halves instead, reducing the expansion depth from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$ but this is more difficult to implement and provides little practical benefit in all but the most extreme branching cases.

caller for further operations. And, of course, the message is transmitted over the underlying transport the nature of which is not restricted by this abstraction. The only requirement is that the Message trait can be converted to a representation that the channel can process, which is the reason for the trait in the first place.

The implementation of the offer methods is slightly more involved. Once a message is received from the underlying transport we must determine which branch of the offer to take and convert it to the appropriate message type. We outsource the decision to a function we receive as an argument called the *picker*. We find that in our particular use case, having the capability to differentiate branches based on external context is necessary. This allows us to distinguish the receipt of an expected packet from the error condition when an unexpected packet is received

4.5 Establishing a Connection

A TCP connection is established via a three-way handshake as described in Section 3. We define the ServerSystemSessionType to describe creation of the server socket (receipt of Open from the server user), creating the internal state (the "TCB"; §A.1), waiting for a SYN from the client, and generating the SYN-ACK segment, corresponding to the transition through the LISTEN state of Figure 1 into the SYN RCVD state:

```
pub type ServerSystemSessionType = St![
    (RoleServerUser & Open).
    (RoleServerUser + TcbCreated).
    (RoleClientSystem & Syn).
    (RoleClientSystem + SynAck).
    ServerSystemSynRcvd
];
```

The ServerSystemSynRcvd type describes the SYN RCVD state, with branches indicating the transition to the ESTAB state in ServerSystemCommLoop if the received ACK is acceptable or closing the connection if not.

```
Rec!(pub ServerSystemSynRcvd, [
    (RoleClientSystem & {
        Ack. // acceptable (i.e., matches the SYN-ACK sent)
            (RoleServerUser + Connected).
            ServerSystemCommLoop,
        Ack. // unacceptable
            (RoleClientSystem + {
                Ack.ServerSystemSynRcvd, Rst.(RoleServerUser + Close).end
            })
     })
]);
```

The implementation of three-way handshake is further described in Appendix A.1.

4.6 Data Transmission and Re-transmission

When a TCP segment goes unacknowledged for a certain amount of time, it is retransmitted. There are two implementation choices that could be made here: incorporate timeouts into the type system, or leave them out and instead signal session type transitions using external timeouts. The session type theory we are using does not have a notion of timeouts, nor does any session type work containing timeouts [1, 2, 5] have the ability to model the operations needed for TCP timeouts. Hence, we opt to emulate timeouts by introducing a virtual message type and adding it as another branch to the offer session type. In this branch, we continue with a select operation, retransmitting an ACK message and then recursively receiving the next message. The `offer` method on the network channel now accepts another argument, specifying the timeout duration or `None` if no timeout is desired. If the retransmission queue is empty, no timeout should be employed as we run into an issue if it expires – the session type requires a segment to be sent, but there is nothing to send. Further details around data transmission are in Appendix A.2.

4.7 Closing the connection

Closing a TCP connection is a two-step process usually combined into a three-way handshake, as shown in the lower half of Figure 1. Each direction of the stream can be closed independently by sending a segment with the FIN bit set. The Server System session type describes receiving a FIN first and then deciding to close eventually, after allowing the user to send more data using the `ServerSystemCloseWait` session type:

```
Rec!(pub ServerSystemCloseWait, [
    (RoleServerUser & {
        Data.
            (RoleClientSystem + Ack).
            (RoleClientSystem & Ack /* empty ack */).
            ServerSystemCloseWait,
        Close.
            (RoleClientSystem + FinAck).
            (RoleClientSystem & Ack).
            end
    })
]);
```

The case where the server closes first is handled by the `ServerSystemFinWait1` type:

```
pub type ServerSystemFinWait1 = St![
    (RoleClientSystem & {
        Ack. // ACK of FIN
            ServerSystemFinWait2,
        FinAck. // FIN and ACK of our FIN at the same time
            (RoleClientSystem + Ack).
            end
    })];
```

The branch in the `ServerSystemFinWait1` type represents the ways in which the closing handshake can proceed after sending a FIN to close the connection and entering into the FINWAIT-1 state (see Figure 1): either a segment containing an ACK is received causing the system to transition to FINWAIT-2, waiting for a segment containing a FIN indicating that the peer has also finished; or a segment with both FIN and ACK is received causing the final ACK to be sent and terminating the connection via the implied CLOSING and TIME-WAIT states. The `ServerSystemFinWait2` implementation is analogous, but elided due to space constraints.

Finally, in a full TCP implementation, a "simultaneous close" situation can occur where both peers decide to close at the same time. This is not handled by our implementation as it is rarely used and does not fit with the call-and-response style of interaction we model – there is no opportunity for the server to decide to close while waiting for the client.

5 Evaluation

To evaluate our Server System component, we have implemented a simple echo server in the Server User. Every piece of data it receives from the system is split into lines, each line is reversed and then sent back.

The functionality of the server tested is as follows:

Establishing a connection by running netcat and connecting to the server.

Exchanging data with the client by typing in messages manually.

Initiating connection close by sending an empty line to the server. The server user has been programmed to close the connection if an empty line is received.

Responding to connection close by typing `^C` which causes netcat to close the socket and therefore send a FIN to the server.

Correctly handling a FIN-ACK response to a FIN by piping an empty line immediately followed by EOF to netcat. In this situation netcat sends the empty line but does not shutdown the socket immediately. Instead it waits for the server to send a FIN-ACK and then sends a FIN-ACK in response.

We tested our TCP implementation primarily against the Linux kernel TCP stack, running our program and connecting to it using a Linux user-space TCP client (netcat). We have used Scapy [32], a packet manipulation framework, to emulate a misbehaving TCP client or network and evaluate the behaviour of our server in response to this. This included sending packets with invalid sequence numbers, invalid acknowledgement numbers, spurious retransmission or overlapping segments. Finally, we have tested our implementation against the Linux kernel TCP stack with the addition of simulated network errors using the `netem` module to introduce packet loss, delay and reordering. Our test script configures the `TCP_NODELAY` option on the socket and sends messages in a loop with a small delay between them. This ensures that the client sends a lot of small packets to observe the effect of packet loss and reordering. The received data was then compared to the expected output. In all cases, we utilised a packet sniffer to monitor the communication.

All of the presented test cases were found to be handled correctly provided the server is in the ESTABLISHED state. During the opening three-way handshake, after the initial SYN segment, handling is robust as well. We found that the server can handle packet loss and reordering errors and the connection can recover once the impairments are lifted. The server does not cache out of order segments in the receive window affecting performance, since these segments will need to be re-transmitted, but not correctness. This is a limitation inherent in using synchronous session types to model an synchronous protocol that permits reordering and packet loss, and suggests future work to extend the modelling approach.

6 Related Work and Conclusion

Network protocols have been used as examples for various session type theories. The main protocols used as a demonstration in many works are SMTP [3, 6, 17, 18, 19, 20, 21] and POP3 [10, 24]. Both of these protocols are application-layer protocols. Due to this, models of SMTP and POP3 can assume the guarantees provided by the underlying transport layer protocol – in most cases this is TCP. Specifically, any faults, retransmissions and packet re-orderings are handled by the transport layer. In addition to this, these works do implement or model the protocols exactly from the specification with some works only implementing SMTP partially. The specific challenges presented by the network link are not considered and the communication channel is considered only in an abstract manner. This also means that, unlike our implementation of TCP, the works are not shown to connect or work with existing protocol stacks, such as the kernel. To our best knowledge, ours is the first work to consider the implementation of transport layer protocols from their specification using session types.

In this paper, we have modelled TCP of the Internet protocol suite [7] using MPST [31] and implemented a proof of concept in the Rust programming language, leveraging the Rust type system and borrow checker to verify that the implementation complies with the session type. We have successfully tested our implementation using manual testing against the Linux kernel TCP stack as well as manually constructed TCP segments. In future work, we aim to address limitations of our implementation such as a lack of timeouts in the type system and the synchronous nature of our implementation. We additionally aim to model important aspects of the protocol such as congestion control in the future.

References

[1] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida & Fangyi Zhou (2022): *Generalised Multiparty Session Types with Crash-Stop Failures*. In Bartek Klin, Slawomir Lasota & Anca Muscholl, editors: *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, *LIPIcs* 243, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:25, doi:10.4230/LIPICS.CONCUR.2022.35.

[2] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos & Nobuko Yoshida (2019): *Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes*. In Luís Caires, editor: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, *Lecture Notes in Computer Science* 11423, Springer, pp. 583–610, doi:10.1007/978-3-030-17184-1_21.

[3] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos & Nobuko Yoshida (2019): *Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes*. In Luís Caires, editor: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, *Lecture Notes in Computer Science* 11423, Springer, pp. 583–610, doi:10.1007/978-3-030-17184-1_21.

[4] Scott O. Bradner (1996): *The Internet Standards Process – Revision 3*. RFC 2026, doi:10.17487/RFC2026. Available at `https://www.rfc-editor.org/info/rfc2026`.

[5] Matthew Alan Le Brun & Ornela Dardha (2023): *MAGπ: Types for Failure-Prone Communication*. In Thomas Wies, editor: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, *Lecture Notes in Computer Science* 13990, Springer, pp. 363–391, doi:10.1007/978-3-031-30044-8_14.

[6] Christian Bartolo Burlò, Adrian Francalanza & Alceste Scalas (2021): *On the Monitorability of Session Types, in Theory and Practice (Artifact)*. *Dagstuhl Artifacts Ser.* 7(2), pp. 02:1–02:3, doi:10.4230/DARTS.7.2.2.

[7] Wesley Eddy (2022): *Transmission Control Protocol (TCP)*. RFC 9293, doi:10.17487/RFC9293. Available at `https://www.rfc-editor.org/info/rfc9293`.

[8] Heather Flanagan (2019): *Fifty Years of RFCs*. RFC 8700, doi:10.17487/RFC8700. Available at `https://www.rfc-editor.org/info/rfc8700`.

[9] Simon Fowler (2016): *An Erlang Implementation of Multiparty Session Actors*. In Massimo Bartoletti, Ludovic Henrio, Sophia Knight & Hugo Torres Vieira, editors: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, *EPTCS* 223, pp. 36–50, doi:10.4204/EPTCS.223.3.

[10] Simon Gay, Vasco Vasconcelos & António Ravara (2003): *Session Types for Inter-Process Communication*. Available at `https://www.dcs.gla.ac.uk/~simon/publications/TR-2003-133.pdf`.

[11] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–138, doi:10.1007/BFb0053567.

[12] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, Association for Computing Machinery, New York, NY, USA, pp. 273–284, doi:10.1145/1328438.1328472.

[13] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In Jan Vitek, editor: *ECOOP 2008 – Object-Oriented Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 516–541, doi:10.1007/978-3-540-70592-5_22.

[14] Thomas Bracht Laumann Jespersen, Philip Munksgaard & Ken Friis Larsen (2015): *Session Types for Rust*. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, Association for Computing Machinery, New York, NY, USA, pp. 13–22, doi:10.1145/2808098.2808100.

[15] Wen Kokke (2019): *Rusty Variation: Deadlock-free Sessions with Failure in Rust*. *Electronic Proceedings in Theoretical Computer Science* 304, pp. 48–60, doi:10.4204/eptcs.304.4.

[16] Wen Kokke & Ornela Dardha (2021): *Deadlock-free session types in linear Haskell*. In: *Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021*, ACM, pp. 1–13, doi:10.1145/3471874.3472979.

[17] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2016): *Typechecking protocols with Mungo and StMungo*. In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, ACM, pp. 146–159, doi:10.1145/2967973.2968595.

[18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2018): *Typechecking protocols with Mungo and StMungo: A session type toolchain for Java*. *Sci. Comput. Program.* 155, pp. 52–75, doi:10.1016/J.SCICO.2017.10.006.

[19] Nicolas Lagaillardie, Rumyana Neykova & Nobuko Yoshida (2022): *Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types (Artifact)*. *Dagstuhl Artifacts Ser.* 8(2), pp. 09:1–09:16, doi:10.4230/DARTS.8.2.9.

[20] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In Jan Vitek, editor: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, *Lecture Notes in Computer Science* 9032, Springer, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[21] Sam Lindley & J. Garrett Morris (2016): *Talking bananas: structural recursion for session types*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 434–447, doi:10.1145/2951913.2951921.

[22] Stephen McQuistin, Vivian Band, Dejice Jacob & Colin Perkins (2020): *Parsing Protocol Standards to Parse Standard Protocols*. In: *Proceedings of the Applied Networking Research Workshop*, Association for Computing Machinery, New York, NY, USA, p. 25–31, doi:10.1145/3404868.3406671.

[23] Stephen McQuistin, Vivian Band, Dejice Jacob & Colin Perkins (2021): *Investigating Automatic Code Generation for Network Packet Parsing*. In: *Proceedings of the IFIP Networking Conference*, pp. 1–9, doi:10.23919/IFIPNetworking52078.2021.9472829.

[24] Matthias Neubauer & Peter Thiemann (2004): *An Implementation of Session Types*. In Bharat Jayaraman, editor: *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, *Lecture Notes in Computer Science* 3057, Springer, pp. 56–70, doi:10.1007/978-3-540-24836-1_5.

[25] Nicholas Ng & Nobuko Yoshida (2016): *Static deadlock detection for concurrent go by global session graph synthesis*. In Ayal Zaks & Manuel V. Hermenegildo, editors: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, ACM, pp. 174–184, doi:10.1145/2892208.2892232.

[26] Nicholas Ng, Nobuko Yoshida & Kohei Honda (2012): *Multiparty Session C: Safe Parallel Programming with Message Optimisation*. In Carlo A. Furia & Sebastian Nanz, editors: *Objects, Models, Components, Patterns*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 202–218, doi:10.1007/978-3-642-30561-0_15.

[27] Luca Padovani (2017): *A simple library implementation of binary sessions*. *J. Funct. Program.* 27, p. e4, doi:10.1017/S0956796816000289.

[28] Vern Paxson (1997): *Automated packet trace analysis of TCP implementations*. In: *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 167–179, doi:10.1145/263105.263160.

[29] Riccardo Pucella & Jesse A. Tov (2008): *Haskell session types with (almost) no class*. In Andy Gill, editor: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, ACM, pp. 25–36, doi:10.1145/1411286.1411290.

[30] Pete Resnick (2014): *On Consensus and Humming in the IETF*. RFC 7282, doi:10.17487/RFC7282. Available at `https://www.rfc-editor.org/info/rfc7282`.

[31] Alceste Scalas & Nobuko Yoshida (2019): *Less is More: Multiparty Session Types Revisited*. *Proc. ACM Program. Lang.* 3(POPL), doi:10.1145/3290343.

[32] Scapy community: *Scapy*. `https://scapy.net/`.

A Appendix

A.1 Three-way handshake

The session type and the TCP state machine are initiated in the CLOSED state:

```
let st = ServerSystemSessionType::new();
let tcp = TcpClosed::new();
```

The *user* role calls the `Open` method and a TCB is created. This message is received using `offerone` by the system role:

```
let (_open, st) = system_user_channel.offer_one(st);
```

The system now transitions to the LISTEN state, waiting for a connection establishment to initiate, and sends a `TcbCreated` in response:

```
let tcp: TcpListen = tcp.open(LocalAddr { /* ... */ });
let st = system_user_channel.select_one(st, TcbCreated(()));
```

The next steps are to wait for a SYN segment from the network and respond with a SYN ACK segment. Once a SYN segment is received we transition to the SYN-RCVD state:

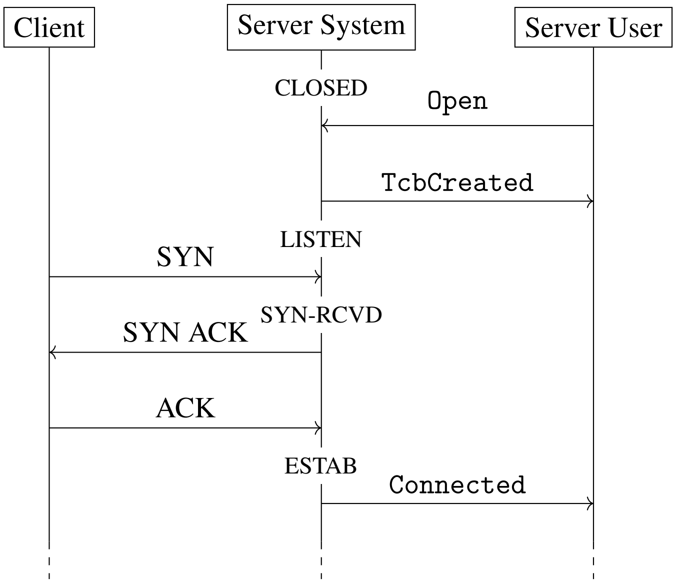


Figure 2: TCP three-way Handshake with all roles.

```
let (addr, syn, st) = net_channel.offer_one_with_addr(st, &tcp);

let (mut tcp /* Tcp<SynRcvd> */, synack) = tcp.recv_syn(addr, &syn);
let mut syn_rcvd = net_channel.select_one(st, addr, synack);
```

The recursive SynRcvd session type handles unacceptable acknowledgements of SYN ACK segments which need to be responded to with an ACK with the potential of a connection reset:

```
let (mut tcp, st) = loop {
    let st = syn_rcvd.inner();
    let tcp_for_picker = tcp.for_picker();
```

The offer_two_filtered method can now be called on the network channel. This method takes the session type, a picker function, and a channel filter:

```
    match net_channel.offer_two_filtered(
        st,
        |packet| match tcp_for_picker.acceptable(&packet) {
            ReactionInner::Acceptable(_, _) => Branch::Left(
                packet.into()),
            _ => Branch::Right(packet.into()),
        },
        &tcp,
    ) {
```

Note that the picker determines which branch to take based on the TCP state machine.

The left branch of the session type corresponds to an acceptable ACK segment:

```
        Branch::Left((acceptable, st)) => {
            let tcp: Tcp<Established> = tcp
```

```
                .recv_ack(&acceptable)
                .empty_acceptable()
                .expect("First␣ACK␣must␣be␣empty");
            break (tcp, st);
        }
```

However, if the ACK is not acceptable, the right branch is taken – an ACK is either sent back and the system waits for another ACK:

```
        Branch::Right((unacceptable, st)) => {
            let remote_addr = tcp.remote_addr();
            match tcp.recv_ack(&unacceptable) {
                Reaction::NotAcceptable(tcp2, Some(response)) => {
                    let st = net_channel.select_left(
                        st, tcp2.remote_addr(), response);
                    syn_rcvd = st;
                    tcp = tcp2;
                    continue;
                }
```

Alternatively, an RST, notifying the user that the connection is being reset:

```
                Reaction::Reset(Some(rst)) => {
                    let st = net_channel.select_right(
                        st, remote_addr, rst);
                    let end = system_user_channel.select_one(
                        st, Close(()));
                    net_channel.close(end);
                    system_user_channel.close(end);
                    return;
                }
```

Finally, once out of the loop, the implementation is in the ESTAB state and the system notifies the user that the connection is established:

```
let mut recursive = system_user_channel.select_one(st, Connected(()));
info!("established");
```

This concludes the implementation of the three-way handshake.

A.2 Exchanging data

The main loop of the implementation waits to receive a segment (using an `Offer` session type) and branches based on their type and whether it is acceptable or not.

```
Rec!(pub ServerSystemCommLoop, [
    (RoleClientSystem & {
```

Acceptable with payload where an acceptable segment is received and there is data present:

```
Ack.
    (RoleClientSystem + Ack /* empty */).
    (RoleServerUser + Data).
    (RoleServerUser & {
        Data.
            (RoleClientSystem + Ack /* with data */).
            ServerSystemCommLoop,
        Close.
            (RoleClientSystem + FinAck).
            ServerSystemFinWait1
    }),
```

Initially, data acknowledgement is accomplished using an empty ACK segment. The data contained within the ACK segment is then transmitted to the server user within a message of type `Data`. The user has the option to respond by sending back a message, leading to the transmission of an ACK with payload. Alternatively, the user may choose to initiate the closure of the connection, resulting in the transmission of a FIN ACK.

Acceptable without payload In the case where these segments are acknowledgements of previously sent segments, we pass the ACK to the TCP state machine to update the state and update the retransmission queue:

```
Ack.ServerSystemCommLoop,
```

FIN ACK The peer has initiated closing the connection. In this case, the TCP state machine will transition from ESTABLISHED to the CLOSE-WAIT state. Note that due to the absence of timeouts in the type system, the timeout in the close-wait state is implemented outside the session typed state machine.

```
FinAck.
    (RoleClientSystem + Ack /* we ACK the FIN */).
    (RoleServerUser + Close).
    ServerSystemCloseWait,
```

Unacceptable A segment where the sequence numbers are not acceptable has been received. The server will respond with an ACK which serves to inform the peer about the current receive window start and length [7].

```
Ack.
    (RoleClientSystem + Ack).
    ServerSystemCommLoop,
```

```
    })
]);
```