University of Glasgow | School of Computing Science

A Rust implementation of TCP utilising multiparty session types

Samuel Čavoj

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8RZ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

September 1, 2023

Abstract

The TCP/IP protocol suite forms the Internet's backbone, enabling seamless communication across devices. One of the protocols is TCP which bridges the gap between the need for reliable data transfer and the reality of unreliable packet-switched networks. TCP, like many other protocols, is specified in an RFC document published by the IETF. The RFCs are written mostly in informal English and are often ambiguous which can lead to interoperability issues.

Session types are a typing discipline which offers a formal approach to protocol descriptions. Implementations can be checked for compliance with a session type using a type checker. The type system of the Rust programming language is strong enough to represent session types natively. Multiparty session types can represent communication among more than two parties.

This work first introduces the relevant background information about TCP and session types. It discusses how multiparty session types can be represented in Rust and details an implementation of a subset of TCP using them. This TCP implementation is capable of basic operation including establishing a connection, transferring data bidirectionally and closing the connection gracefully. The correctness of the implementation is tested against the Linux TCP stack, its limitations are identified and future work to overcome some of the limitations is proposed.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format.

Samuel Čavoj

Acknowledgements

I would like to thank my supervisor Dr. Ornela Dardha for her guidance and supervision of the dissertation, Ivan Nikitin for his help with the project and for providing a starting point for me to bounce off, and finally my girlfriend for her support, help with my hopeless time management skills, and patience which was surely tested while listening to me ramble on about yet another computer science thing.

Contents

Chapter 1: Introduction

Network protocols 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 TCP/IP, is defined in a series of RFCs published by the IETF. Specifically, TCP is defined in RFC 9293 [3].

The RFCs are written mostly in informal English and are intended to be read by humans. However, they are also used by software developers to implement the protocols. The specifications are often ambiguous and incomplete, which can lead to interoperability issues between different implementations or security issues [1, 11]. Furthermore these protocols are deployed on a massive scale and any change to the specification must be carefully considered and tested to ensure that it does not break existing implementations which cannot be easily updated. Any updates to the specification need to take into account behaviour of existing systems carefully.[1] This is especially true for TCP which is a very old protocol and has been in use since the early days of the Internet.

Session types

Session types 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 a number of desirable properties. They have been an active area of research since the late 1990s [5] and have been implemented in a number of programming languages such as C [10], Java [7] and Rust [9, 8]. An introduction to session types can be found in section 2.2.

Rust

Rust is a programming language well suited for the development of fast and efficient programs. It is commonly used for systems programming and focuses on safety. Rust has strict rules and if they are followed the resulting program is guaranteed to be memory- and thread-safe. These rules are enforced via static analysis techniques such as the borrow checker, in which ownership of values is tracked and the creation of references is subject to conditions. Combined with an affine type system and algebraic data types, this results in a powerful language which is quite practical for real-world uses.

Thanks to Rust's affine type system, it is possible to implement session types directly using Rust and without the need of an external tool for verification – the verification can be performed by the Rust compiler [8]. Affine type systems are characterised by the fact that values can be used at most once. Without this requirement, it is not possible to rely on the type system to verify session types.

[1] For an example, see the sections of RFC 9293 on the urgent pointer [3, § 3.8.5] or RFC 6093 [4].

This is because the value associated with the session type could be duplicated and re-used, bypassing the session type.

Goals

The goal of this project is to implement the TCP protocol in the Rust programming language while using session types to describe 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 is utilised to verify that the implementation of the protocol is correct in terms of the types of messages exchanged and the order in which they are exchanged, at least as long as the session type itself is correct. Session types are also used to describe the application interface and hence it is verified that the application uses the TCP implementation correctly.

We would like to accomplish at least the following goals:

1. Model a viable subset of TCP using session types.
2. Write infrastructure required for encoding the session type model into native Rust types in an ergonomic fashion.
3. Implement both the user/TCP interface and the TCP/lower-level interface using this model while adhering to the session type model. This is to be done in a way such that the Rust compiler can detect deviation from the session type.
4. Test the implementation against a real TCP stack.

Dissertation structure

This dissertation is structured as follows: in the Background chapter (2) we introduce the underlying concepts in general terms with little focus on our implementation. In the Implementation chapter (3) we describe the specifics of our Rust implementation and the challenges we faced. In the Evaluation chapter (4) we describe how we tested our implementation and its limitations. Finally, in the Conclusion chapter (5) we summarise our work and discuss possible future directions.

Chapter 2: Background

In this chapter we will introduce the fundamentals of the Transmission Control Protocol and multiparty session types and their representation in Rust.

2.1 Transmission Control Protocol

The Transmission Control Protocol (TCP) is one of the core protocols of the Internet. It provides reliable delivery of a stream of bytes between two applications. It utilises the Internet Protocol (IP) to deliver the data and detects and corrects any transmission errors. Furthermore TCP provides congestion control to maximise utilisation of the underlying network link and fairness among different connections. It is a connection-oriented protocol, meaning that a connection must first be established before data can be transferred. This is done using a three-way handshake which also serves as a security mechanism to prevent spoofing of the source address. Data can be transferred in both directions simultaneously. In this description the terms *sender* and *receiver* are to be understood as the sending and receiving components of the TCP implementation on either end of the connection.

The service provided to TCP by the lower layer (generally IP) is an unreliable datagram service. As such packets can be lost, duplicated or they can arrive in a different order than they were sent. To provide a reliable stream of bytes service, the sender must segment the stream into packets of a limited size and transmit them over the network while the receiver must reassemble the packets into the original stream after confirming that all previous data was correctly received. If the sender detects a lost packet, either by way of a timer expiring before an acknowledgement is received or by receiving a duplicate acknowledgement, it will retransmit the packet. The receiver will discard duplicate packets and will send back acknowledgements for all data that was received correctly.

A slightly more detailed description of the protocol follows, but refer to the RFC [3] for a complete specification.

2.1.1 Message format

Each TCP packet consists of a header and a payload. Both of these are then encapsulated within the payload of an IP packet providing separation between the layers. The header contains the source and destination port numbers used to identify a connection among all connections between the same pair of endpoints, sequence and acknowledgement numbers used to indicate which part of the stream is contained in the message and to detect lost or duplicated packets, the window size used for flow control, control bits indicating the purpose of the message and special circumstances, and a checksum used to detect transmission errors. The optional payload contains the data being sent.

The control bits defined by the protocol are as follows:

SYN Synchronise sequence numbers. Used only in the three-way handshake to initialise a connection.

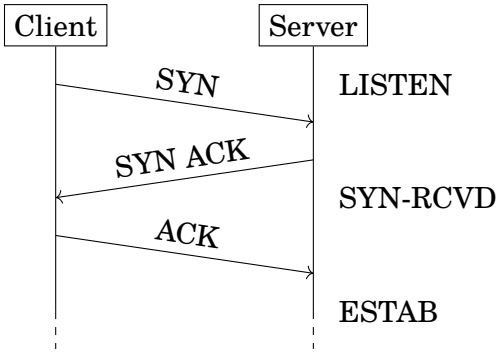

Figure 2.1: TCP three-way Handshake

FIN Finish. Indicates that the sender has no more data to send. Once both sides have received and acknowledged a FIN message, the connection is closed.

RST Reset. Used to abnormally terminate a connection.

PSH Push. Indicates that the data should be delivered to the application as soon as possible.

ACK Acknowledgement. The acknowledgement number is valid and contains the sequence number of the next byte expected by the sender.

URG Urgent. The urgent pointer is valid.

2.1.2 Three-way handshake

For the purposes of this description, we will only consider the case where one party is a server listening for incoming connections (passive) and the other is a client initiating the connection (active). The description will be from the perspective of the server.

The server begins in the CLOSED state. Once it is ready to accept connections, it enters the LISTEN state and waits to receive the first packet from the client. This packet must have the SYN control bit set and the sequence number set to an unpredictable value. The server then must acknowledge the SYN by sending a SYN ACK packet with the acknowledgement number set to the client's sequence number plus one[1] and the sequence number set to an unpredictable value of its own. The server then enters the SYN-RECEIVED state. Finally, the client must acknowledge the server's SYN in an analogous way. Once the server receives the acknowledgement, it enters the ESTABLISHED state and the connection is ready to be used. At this point both sides know the other's sequence number and can begin sending data.

This procedure is known as the three-way handshake, since three messages were exchanged. See fig. 2.1 for a visualisation.

[1]Normally the sequence number is increased by the length of the payload, but both the SYN and FIN flags take up one byte in the sequence number space, SYN being virtually the first byte and FIN being virtually the last byte. This is done so that they need to be acknowledged the same way data bytes are.

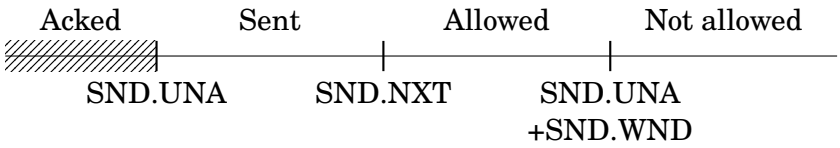

Figure 2.2: TCP sender sliding window

2.1.3 Normal operation

Once the connection is established and both sides are in the ESTABLISHED state, data can be transferred in both directions. The implementation must keep track of several state variables to operate. The most important of these are SND.UNA, SND.NXT and SND.WND which keep track of the outgoing data stream and RCV.NXT and RCV.WND which keep track of the incoming data stream.

SND.UNA Sequence number of the first byte of sent data that has not yet been acknowledged by the peer.

SND.NXT Sequence number of the next byte of data to be sent.

SND.WND Size of the receive window advertised by the peer.

RCV.NXT Sequence number of the next byte of data expected to be received from the peer.

RCV.WND Size of the receive window advertised to the peer.

When transmitting a segment (optionally containing a data payload) its sequence number is chosen to be SND.NXT. This informs the receiver of the position of the segment within the stream and allows it to detect lost or duplicated segments. The acknowledgement number is set to RCV.NXT and it denotes the sequence numbers which the sender has received and processed, relating to the data stream in the opposite direction.

The sender must keep data available for retransmission until they are acknowledged by the receiver. These are the bytes between SND.UNA and SND.NXT. Bytes before SND.UNA have been acknowledged and no longer need to be kept. Bytes after SND.NXT have not yet been provided by the application, but the space between SND.NXT and SND.UNA + SND.WND is usable for future transmissions. The space past SND.UNA + SND.WND should not be used until an acknowledgement is received which shifts this further. See fig. 2.2 for an illustration. If data have been sent but not yet acknowledged and the retransmission timer expires, the data should be sent again and the timer reset. The duration of the retransmission timer is known as the retransmission timeout (RTO) and is to be determined dynamically in order to reflect network conditions.

Upon receiving a segment it is determined whether the sequence number (SEG.SEQ) is acceptable. Broadly speaking, segments whose beginning (SEG.SEQ) or end (SEG.SEQ + SEG.LEN) fall into the current receive window are deemed acceptable. The segments are then processed in order of increasing sequence numbers or cached if there remains a gap before them in the receive window. If an unacceptable segment is received, a segment with no data and the ACK bit set is transmitted in response. This carries the current value of RCV.NXT and is useful for the peer if previous acknowledgements were lost.

TCP implementations typically maintain send and receive buffers for the application to append to and read from. This isolates the application from the concerns of segmentation and presents a simple stream of bytes interface. The TCP sender reads from the send buffer, creates segments of appropriate size and sends them when the receiver's window and the congestion control algorithm allow for it. Similarly, the TCP receiver writes to the receive buffer when data arrives from the network. These buffers are of finite size and may fill up. The receive buffer state regulates the size of the receive window advertised to the peer.

2.1.4 Closing the connection

Once the sender determines there is no more data to send (usually by receiving a command from the application such as `shutdown(2)` on Unix) it signals this to the peer by sending a FIN packet. From this point on it is not permitted to send data in any following messages, which will therefore be only acknowledgements of received data. Once FIN packets are sent in both directions of the connection and both are acknowledged the connection is closed. If a sender sends a FIN without having received one prior to that and later acknowledges the second incoming FIN, it must transition to a TIME-WAIT state and wait until a timer expires to ensure that the peer has received the acknowledgement. Once the timer expires it can transition back to the CLOSED state and open new connections.

2.2 Session types

Session types are a typing discipline aiming to formally 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. A simple example of a binary session type is the following: !`int`.?`str`.`end`. This describes a protocol where the first action is to send a message of type `int` followed by receiving a message of type `str` and finally closing the channel. The send and receive operations are denoted by ! and ? respectively and the period denotes sequencing. The `end` type denotes a closed channel with which no more actions can be performed. In order for the communication to be safe, the participant on the other end of the channel would have the session type ?`int`.!`str`.`end`. These two session types are *dual* to each other and describe the same protocol from the two perspectives of each of the two participants.

Having only send and receive actions is quite limiting in practice and two more actions are usually defined, namely selection and offer. These are used to branch the protocol into multiple paths. The "select" action is denoted by $\oplus$ and allows the participant to choose one of the branches and send a message of the corresponding type. The dual of this is the "offer" action denoted by & which simply means that a message of any of the offered types can be received and the decision of which branch was taken is made externally by the peer. Each branch specifies a message type and a continuation session type. The branches may also have textual labels but these have no formal function.

A simple binary example featuring selection and offer are the following dual session types:

$$\begin{aligned} S &= \oplus\,\{l_1(\texttt{int}).\texttt{end}, l_2(\texttt{str}).\texttt{end}\}, \\ \overline{S} &= \&\,\{l_1(\texttt{int}).\texttt{end}, l_2(\texttt{str}).\texttt{end}\}. \end{aligned} \tag{2.1}$$

In this example the participant described by session type S can decide to take either of the two available branches. Its peer (described by $\overline{S}$) receives the message and is forced to take the corresponding branch. Note that the concept of choice in case of selection is entirely up to the implementation fulfilling the session type. The session type only serves as a tool to verify that one of the branches was followed but has no say in which one it is. On the other hand, in case of offer, the implementation is entirely dependent on what arrives from the peer over the channel.

Traditionally, one would distinguish the send action (!) from the selection action ($\oplus$), but send is essentially the same as a selection with only one branch, so in this work only selection will be used to describe sending. The same reasoning can be applied to the receive action (?) and the offer action (&).

In addition to branching, a notion of repetition is required for a useful protocol description language. It is natural to use recursion for this purpose by binding a name (e.g. `T`) to a part of the session type and referring to it later. To do this the μ operator is defined with a syntax akin to functions in lambda calculus. A simple example of a recursive session type which can send a message of type `M1` arbitrary number of times and finally send one message of type `M2` and terminate is shown below in eq. (2.2).

$$S = \mu \texttt{T}.\ \oplus \{l_1(\texttt{M1}).\texttt{T}, l_2(\texttt{M2}).\texttt{end}\} \tag{2.2}$$

So far we have been talking about *binary* session types which can only describe protocols between two participants [5]. However, the theory was later extended to multiparty session types (MPST) which can describe protocols between any number of participants [6]. Initially this was done using a single type which described a global view of the protocol (the global type) which could then be projected onto views of the individual roles. In later works this was discovered to be needlessly complex and not powerful enough to describe many protocols. The new "Less is more" [12] approach forgoes the global type and instead specifies a session type for each role individually. Various properties can then be analysed given a *typing context* – a set of session types for the individual roles in the protocol. In this project Less is more multiparty session types are used to describe TCP. See chapter 3 for details about the implementation and which roles are considered.

In MPST, there is no longer a concept of duality so checking safety is more involved. Leveraging the Rust compiler to verify safety or other properties would be an interesting research avenue, but is not the object of this project.

A practical introduction to Less is more MPST is that in contrast with binary session types, each operation requires a target role. Therefore each operation gets an additional attribute, the peer role it is interacting with. As an example, let us consider the typing context Γ in eq. (2.3) having two participants with roles `R1` and `R2`. Participant `R1` will send any number of messages of type `M1` to `R2` via the l_1 branch and finally send a single message of type `M2` before reaching `end` via the l_2 branch. The session type $\mathbb{s}$[`R1`] below describes this behaviour and $\mathbb{s}$[`R2`] describes a corresponding session type for the other role. This is essentially the same example as in eq. (2.2) but adapted to MPST. While the types might still seem dual to each other, the concept of duality is no longer defined in this context and it breaks down when more than two roles are present.

$$\begin{aligned}\Gamma \;=\; & \mathtt{s}[\mathtt{R1}] : \mu \mathtt{T}.\mathtt{R2} \oplus \{l_1(\mathtt{M1}).\mathtt{T}, l_2(\mathtt{M2}).\mathtt{end}\}, \\ & \mathtt{s}[\mathtt{R2}] : \mu \mathtt{T}.\mathtt{R1} \,\&\, \{l_1(\mathtt{M1}).\mathtt{T}, l_2(\mathtt{M2}).\mathtt{end}\}.\end{aligned} \tag{2.3}$$

Naturally, the constructs described in this chapter can be combined and nested arbitrarily. In the following chapters we delve into how TCP can be described using this concept, how the Rust compiler can be utilised to check these types and what limitations we run into.

More powerful session type theories exist and deal with buffering, lossy network channels, timeouts, asynchronous communication, and many other ideas [2] but we will not be exploring these in this project.

Chapter 3: Implementation

We have implemented the basic functionality of the TCP protocol while modelling both the network and the application interface using session types. Under the *Less is more* formalisation of multiparty session types [12] the roles we are considering are 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 application interface and do not pass over the network. These would normally be implemented as function calls but for the sake of clarity we have chosen to implement them as messages passed over a channel between two threads.

The two threads (representing the user and the system) each have a session type which prescribes their behaviour relative to the other roles. The third (*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.

3.1 Multiparty session types in Rust

This section introduces the concepts behind the implementation of session types in Rust, but omits specific TCP-related details which are described later in section 3.3.

3.1.1 Defining session types

The basic building blocks of this 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 the messages exchanged, and the continuation of the session. In addition, the `End` struct is also an `Action` and represents the end session type.

Taking a closer look at the `OfferTwo` struct, we note that it has five type parameters. The first is the peer role, the next two are the types of the 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. The struct is zero-sized containing only a phantom data field to use up the type parameters.

```
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 `SelectTwo` struct has the same type parameters and is also zero-sized. The semantics are, of course, different. Finally, the non-branching actions `OfferOne` and `SelectOne` have only three type parameters: the peer role, the message type, and the continuation type. Selections and offers with more than two branches are implemented by nesting the two-branch actions. This is discussed in more detail in section 3.1.2.

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
]
```

In this language the `&` operator represents an offer and the `+` operator represents a selection. An example with branching might look like the following:

```
St![(Client & {
    Request1.(Client + Response1).end,
    Request2.(Client + Response2).end
})]
```

This type is expanded to the following:

```
OfferTwo<Client, Request1, Request2,
    SelectOne<Client, Response1, End>,
    SelectOne<Client, Response2, End>
>;
```

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

3.1.2 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 section 3.1.4.

However, support for more than two branches is required in practice. A simple (but functional) way to do this is to implement `OfferThree`, `OfferFour`, ..., in the same way, along with the code supporting this. This leads to more duplication but does not increase complexity and the usage is very straightforward. It is likely that some of the duplication could be reduced using macros, but this approach was not explored further.

Instead, 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.[1] 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. Let us only consider the offer operation. The inner `OfferTwo` nodes are semantically different from the top-level one or other simple `OfferTwo` session types. Normally, the type embodies a *receive* action on a channel, but in our tree, if the top-level session type is used to receive a message, the nested types must represent something different, as the message has already been received, and it makes no sense to perform another receive using the continuation session type. This leads to the question of what the message type should be for the inner *right* branches.

In the semantics we chose, the topmost `OfferTwo` is used to perform the receive action on the channel and the virtual nodes are merely a session type path to get to the result. Consider an enum `Either<L, R>` with two variants `Left(L)` and `Right(R)`. We use this as the message type for *right* branches leading to virtual nodes. As an example, consider the following three-way branching using our macro notation:

```
type A = St![(Client & {
    Request1.(Client + Response1).end,
    Request2.(Client + Response2).end,
    Request3.(Client + Response3).end
})]:
```

This is expanded into the following:

```
type A = OfferTwo<Client,
    Request1,
    Either<Request2, Request3>,
    SelectOne<Client, Response1, End>,
    OfferTwo<NestRole,
        Request2,
        Request3,
```

[1] 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.

```
        SelectOne<Client, Response2, End>,
        SelectOne<Client, Response3, End>
    >
>;
```

Upon using the outer `OfferTwo`, depending on the received message, the user obtains either a `Request1` and a normal continuation session type and can proceed normally, or, if the received message is one of the other branches, the received message is the `Either` type and the continuation is another `OfferTwo` with a special role called `NestRole`. This role is used to denote that the action does not represent a receive from a channel, but rather a means to extract the message from the `Either` instance. The output of this is then a `Request2` or `Request3` depending on what is contained within the `Either` and one of the normal continuation session types. In larger trees, this approach can be applied recursively until the bottom of the tree is reached.

In summary this approach allows us to implement branching of arbitrary arity using only the two-way branching constructs. While the expanded type definitions become more complex, this is not a problem in itself as the user does not need to specify them directly, and can use the macro notation instead. A more serious issue is that using the session types becomes more cumbersome. Instead of the ideal case of a single `match` statement with an arm for each branch, the user must use nested `match` statements, one for each level of the tree. Furthermore, we have not implemented multi-way *selection* using this approach, as it was not necessary for the purposes of this project. While it is likely possible, there might be obstacles. This area is something that could be explored in the future.

3.1.3 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`. Furthermore, in this case nothing is stored at all as the session types carry no data and are all zero-sized. Recursion in only the type parameters poses no problem, though a `PhantomData` marker field which uses the type parameter needs to be added.

To use this solution, one needs to define a new struct for every μ operator in the session type. As an example, instead of writing the following (which is not allowed as noted above)

```
type A = SelectOne<_, _, A>;
```

one can write

```
struct A(SelectOne<_, _, A>);
```

or analogous. To then use the resulting session type an extra step must be performed to extract the inner value from the wrapper, though this is not a big issue. In addition to this, for each session type defined in this way, a trait implementation for the `Action` trait is required. This increases the amount of clutter surrounding the definitions, potentially reducing readability. To counteract this we have defined a macro which defines the struct and generates the related `impl Action` block. The example from above would be written as `Rec!(A,SelectOne<_, _, A>)` and this is all that is needed. Still, another downside is that a name for the type is now a requirement and that this cannot be done inline within the definition of another session type.

A different approach to representing recursion is to use another type (e.g. `Rec<S>`) with a type parameter accepting the recursive session type. Then the question becomes how to refer to the type from within when recursion is required as it does not have a name. One option is to use De Bruijn indices which are a way to refer to a variable in a binding context by its position in the stack of bindings. The index can be encoded using Peano numbers in the type system. This approach has been used in the Rust implementation of session types by Jespersen et al. [8]. However, we have not explored this approach further.

3.1.4 Using session types

In the previous sections we discussed how to define various session types, but since they are simply zero-sized structs, there is not much they can do on their own. The key concept is that they can be used as a token, to prove that the section of code which owns a session type instance is entitled to perform the action described by it. As such instances cannot be Cloned or Copied freely, but can be used in conjunction with a *channel* of the correct type. Conversely, a channel cannot be used at all without providing a session type token. In this way the type checker can be utilised to ensure that the channels are used according to the requirements of a session type.

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 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*. On the one hand this makes the abstraction very flexible as the user can implement a context-sensitive picker function, but on the other hand it leaks some of the implementation details. However, we find that in our particular use-case, having the capability to differentiate branches based on external context is necessary. When a packet is received from the network, we might have a branch for when the packet is what we expected and an error branch for unexpected packets caused by network error conditions. These branches must be distinct as the reaction required would be different and as such the continuation session types must be different. It is impossible to distinguish these branches based on the incoming packet alone, the TCP receiver state is required for this. The picker abstraction is a reasonably ergonomic way to implement this setup.

3.2 Network channel implementation

Another prerequisite for a working TCP implementation is a channel (in the session type sense) which can be used to send and receive raw IP packets. As well as this, the operating system on the host machine must be prevented from processing TCP segments and fighting with our own TCP stack.

One way to do this is to use a *raw socket* which allows us to send and receive arbitrary IP packets. Creating a raw socket is a privileged operation and requires the `CAP_NET_RAW` capability on Linux. The socket is created with a specific protocol number which is used to filter incoming packets, in our case TCP with protocol number 6. To avoid conflicts with the kernel TCP stack, we can choose a port number and use the netfilter infrastructure to drop all incoming packets with that port number. This happens before the kernel TCP stack has a chance to process the packet, but it is still delivered to the socket.

Another solution is to use a TUN device. This is a virtual network interface but instead of sending and receiving packets to and from a network card and eventually a physical medium, the packets are sent and received to and from a user-space program. The program can then emulate another host on the network or simply encapsulate the packets and send them over the network (such as a VPN client). For our purposes, the program emulates a host with its own IP address and a route to this address is added on the host machine via this TUN interface. The end result is very similar to the raw socket approach, but there is no need to work around the kernel TCP stack as the packets are never destined for it. In addition, more control over the communication is possible as there is a separate interface over which the packets travel. While to create a new TUN device is a privileged operation, the administrator can create a device beforehand and configure it to be owned by another user. This user can then use the interface freely. A disadvantage is that the implementation is more complex if a proper IP stack is desired.

We have chosen the TUN device approach as it is more flexible and does not require the netfilter hack. For the TUN device and an IP stack we use the `smoltcp` library [14] which provides a user-space networking stack for Rust. We naturally do not use the TCP implementation provided by `smoltcp` and instead build our own atop its IP layer. Once `smoltcp` is configured to use our TUN device and to listen on an IP address, we provide simple `send` and `recv` methods. The former accepts a

destination address and a slice of bytes and sends them as the payload of an IP packet to the address. The latter blocks until a packet is received and returns the source address and the TCP packet, containing the TCP headers and data. Later when the `recv` method is used to implement the `offer` methods of the channel, the picker function can inspect the TCP headers and decide which branch to take, while converting the raw TCP packet to a message type required by the session type.

3.2.1 Channel filtering

A channel should only carry communication between two participants, but in the case of networking, a single IP address can receive packets from any number of hosts, or even multiple TCP connections with the same host. To address this one can simply drop packets which are not part of the current session, by checking the source and destination addresses and ports. However, this cannot be done in a uniform way from the beginning as the remote address and port are not known until the connection is established. Instead we use a *filtering* mechanism which allows for a different filter to be used for each offer operation. The `offer` methods take a reference to an instance of a type implementing the `ChannelFilter` trait. This trait has a single method which takes the source address and the packet and returns a decision whether the packet should be accepted for further processing and should advance the session type or whether it should be dropped without affecting the session type.

3.3 TCP implementation

Utilising the concepts introduced in previous sections we have implemented a rudimentary TCP stack. This section will discuss the implementation in detail. The full source code can be found in appendix A.

3.3.1 Message types

The first step is to define the message types which will be used in the session types. The TCP specification [3] does not differentiate message *types* per se, but rather the logic branches based on the control bits in the header. One option is to use a single message type for all kinds of TCP segments. While this would work, it represents a missed opportunity to use the type system to our advantage. At the other end of the spectrum, we could define a parametrised type with a type parameter for each control bit and have e.g. a `Segment<Syn, Ack>` type. However, Rust does not support variadic generics so this is not possible without a workaround.

Instead we have chosen to define a message type for each reasonable combination of control bits, excluding the URG and PSH bits which are ignored by our implementation. This approach provides the benefits of granular types and does not lead to a combinatorial explosion. Definitions of the message types can be found in listing 3.1. The `smol_message!` macro defines a struct containing the raw TCP packet and implements the `Message` trait for it. It also implements a method to convert the type back to the raw `TcpPacket` and a `From<TcpPacket>` implementation. The `From` implementation checks that the control bits are valid for the message type based on the `smol_message!` macro invocation. Every control bit prefixed with a + must be set and every control bit prefixed with a - must be unset. Bits that are not specified are not checked. This macro allows for a concise definition of the message types. For example attempting to convert a `TcpPacket`

with the SYN and ACK bits set to a Syn message will `panic!` because the ACK must not be set. This is useful to catch correctness errors in the implementation, after all the picker is responsible for ensuring that the bits are correct before converting to a typed message.

```
smol_message!(Syn    { +syn -ack -fin -rst });
smol_message!(SynAck { +syn +ack -fin -rst });
smol_message!(Ack    { -syn +ack -fin -rst });
smol_message!(FinAck { -syn +ack +fin -rst });
smol_message!(Rst    { -syn -ack -fin +rst });
```

Listing 3.1: TCP message types

3.3.2 State machine

Each state of the TCP state machine is represented by its own type and transitions between states are methods on these types. Illegal transitions are not possible as there are no public methods defined to perform them. Many states share the same data and as such they are represented by a single struct with a type parameter for the state. For example the ESTABLISHED state is represented by the `Tcp<Established>` struct. Exceptions to this are the CLOSED and LISTEN states which contain no data, or only the local address and port, respectively. They are represented by their own structs `TcpClosed` and `TcpListen`.

To illustrate the implementation let us consider the ESTABLISHED state and suppose we have an instance of the associated `Tcp<Established>` struct. Some available operations are meant to be called when a segment is received from the network (NET) and some are triggered by the user (USER). See an overview in table 3.1.

	Method	Self	Input	Return type
NET	recv	mut self	&Ack	Reaction<Tcp<Estab>, Tcp<Estab>>
	recv_fin	mut self	&FinAck	Reaction<Tcp<CloseWait>, Tcp<Estab>>
USER	send	&mut self	&[u8]	Ack
	close	mut self		(Tcp<FinWait1>, FinAck)

Table 3.1: Methods of the `Tcp<Established>` type

To interpret the table, we first need to introduce the `Reaction` type. It is an enum representing the possible outcomes of an incoming segment. At the same time it contains the next state of the TCP state machine in case of a successful transition. The `Reaction` enum is defined as follows:

```
#[must_use]
enum Reaction<'a, Ta, Tn> {
    Acceptable(Ta, Option<Ack>, Option<&'a [u8]>),
    NotAcceptable(Tn, Option<Ack>),
    Reset(Option<Rst>),
}
```

The `Ta` and `Tn` type parameters represent the next state in case of a successful transition and an unsuccessful transition, respectively. The `Acceptable` variant is used when the segment is acceptable and the transition is successful. It contains the state machine in the next state, an optional acknowledgement to be sent and a payload to be delivered to the user. The `NotAcceptable` variant means that

the segment is not acceptable. In this case the state machine is returned in the `Tn` state. Optionally an ACK should be sent over the network. Finally the `Reset` variant is used when the segment is not acceptable and the RST bit should be set in the response.

Returning back to table 3.1, the `close` method is to be called when the user wishes to close the connection. It returns a tuple containing the next state of the state machine and a `FinAck` message to be sent to the peer. The `self` parameter is moved into the method, so the user can only interact with the returned instance in the FIN-WAIT-1 state. Combined with Rust's local variable shadowing this is very ergonomic to use. The `recv_fin` method is called when a FIN ACK message is received. If the segment is acceptable, the state machine transitions to the CLOSE-WAIT state and a response ACK is provided as well, though this is not guaranteed by the return type.[2] If the segment is not acceptable, the state machine is kept in the ESTABLISHED state and optionally a response ACK is provided. In case of the `send` method no state transition is possible and only a reference to `self` is required – the caller retains ownership of the instance.

Currently not all of the methods return the `Reaction` type to the caller, some of them handle it internally and `panic!` if they run into an unexpected situation. This is not ideal and should be improved in the future.

The components described in this subsection do not contain any session type related logic and are merely a generic TCP implementation. The session type related code uses these components as a library and implements the session typed interfaces.

3.3.3 Synchronous vs asynchronous

The TCP protocol is inherently asynchronous. Segments can arrive at any time and in any order and, similarly, the user can add data to the send buffer or read from the receive buffer at any time. In addition, retransmission timers can expire. However, the session type abstraction we have chosen is synchronous – the ordering of events is fixed to a large extent. In addition, we chose to model the User/System interface using session types as well, therefore the user's actions are somewhat predetermined.

These issues can be addressed to some extent by using buffering. The network channel implementation uses a TUN device which has a receive buffer in the operating system. Therefore packets *can* arrive at any point in time, regardless of the state of our application and session type. Performing an offer action merely takes the top packet from the buffer and processes it, or blocks if the buffer is empty.

For the purposes of this project we have chosen to adopt a *call and response* style of interaction. In the ESTABLISHED steady state, the server user is waiting (in a call to `offer`) to receive data. It cannot send any data at this point. The server system is waiting to receive a packet from the network. When any packet arrives it is processed and if it is acceptable and contains a payload, the payload is passed to the user. The server user can then send a response which is processed by the server system and sent over the network. This is repeated until the server user decides to close the connection or a FIN packet is received. This pattern is

[2] This is one of the areas of possible improvement in the future, as there is nothing preventing this except the fact that it is more work to implement.

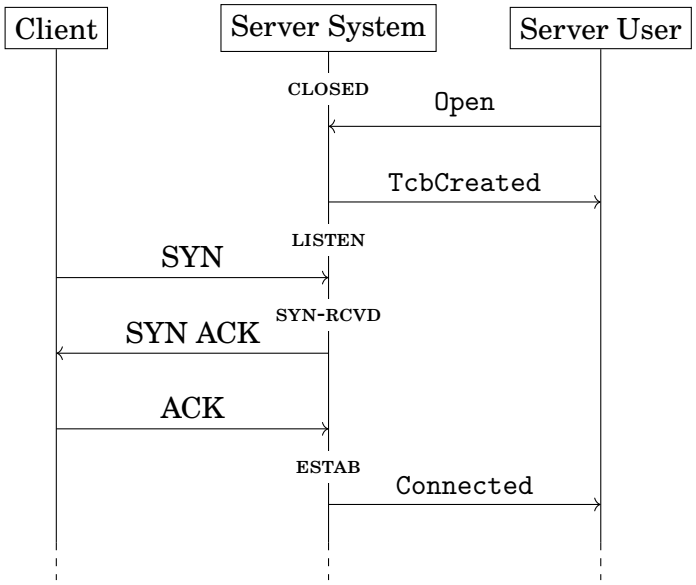

Figure 3.1: TCP three-way Handshake with all roles

exhibited by some application protocols such as HTTP or SMTP (modulo who sends the first message) but is not universal. For example SSH is a protocol which is inherently asynchronous and does not fit this pattern.

3.3.4 Server system implementation

The *server system* is implemented as one of the two threads in the application. It manages one endpoint of the channel connecting it to the *server user* and the `smoltcp` network channel with the *client system*. It owns the TCP state machine. A simplified overview of its operation is that it receives commands from the user, performs actions on the TCP state machine as applicable and sends packets over the network channel. Or, it might receive packets from the network channel, perform actions on the TCP state machine and pass the results to the user. Let us have a closer look at some parts of its functionality.

3.3.5 Three-way handshake

First the three-way handshake must be completed. The session type of the *server system* governing the three-way handshake is shown in listing 3.2 using the macros introduced in section 3.1.1. An illustration of the happy path of the procedure can be seen in fig. 3.1. We will now go through the implementation of this mechanism in a literate programming style as a way to demonstrate the full spectrum of the usage of the session types.

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

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

```
        Ack. // acceptable
            (RoleServerUser + Connected).
            ServerSystemCommLoop,
        Ack. // unacceptable
            (RoleClientSystem + {
                Ack.ServerSystemSynRcvd,
                Rst.(RoleServerUser + Close).end
            })
    })

]);
```

Listing 3.2: Session type of the server system during the three-way handshake

To begin with, we instantiate the session type and the TCP state machine in the CLOSED state.

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

Next the *user* can call the `Open` method. In reality this is a message sent over a channel to the *server system* thread, not a method call. We receive this message using `offer_one`.

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

We then transition to the LISTEN state and send 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. We then transition to the SYN-RCVD state. In the below snippet, the `syn_rcvd` variable contains the recursive session type.

```
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);
```

Finally, we can loop on the recursive `SynRcvd` session type. This handles unacceptable acknowledgements of our SYN ACK which sometimes need to be responded to with an ACK and sometimes the connection is reset. If the SYN ACK is acceptable, we transition to the ESTAB state.

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

We have unwrapped the session type from the containing struct (see section 3.1.3) and made a read-only copy of the TCP state machine. This is because the picker function needs to be able to determine whether a segment is acceptable based on the TCP state, but it must not modify the actual state machine. This was the simplest way to implement this. In essence we are duplicating some work here and a different solution would have to be used if performance was a concern.

We can now call the `offer_two_filtered` method on the network channel. This method takes the session type, a picker function, and a channel filter (see section 3.2.1).[3]

```
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. In this case both branches have the same message type (`Ack`), so that itself is not sufficient to determine the branch. The picker function also takes care of the conversion from the raw packet type into the message type (using the `Into` trait). The picker returns a `Branch` enum which is either `Left` or `Right` depending on which branch needs to be taken. The `offer_two_filtered` method itself also returns a `Branch` enum and it will contain a tuple of the message and the continuation session type token.

The left branch of the session type corresponds to an acceptable ACK segment. We can transition to the ESTAB state and proceed with the continuation session type by breaking out of the loop.

```
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, we take the right branch. We then either send back an ACK and continue the loop to wait 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;
        }
```

…or we send an RST, notify the user that we are closing, and clean up the channels:

```
        Reaction::Reset(Some(rst)) => {
            let st = net_channel.select_right(
```

[3]This is slightly simplified as the real implementation also deals with timeouts.

```
                        st, remote_addr, rst);
                    let end = system_user_channel.select_one(
                        st, Close(()));
                    net_channel.close(end);
                    system_user_channel.close(end);
                    return;
                }
```

We must also enumerate cases which are not possible. Not acceptable with no response, reset with no response and acceptable. If it were acceptable, the picker would have chosen the left branch.

```
                    Reaction::NotAcceptable(_, None) => unreachable!(),
                    Reaction::Reset(None) => unreachable!(),
                    Reaction::Acceptable(_, _, _) => unreachable!(),
                };
            }
        }
};
```

Finally, once we have broken out of the loop, we find ourselves in the ESTAB state and can notify 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. We will not be going into detail on the other parts of the code, but this example could hopefully shed some light on the usage of the primitives we have been introducing up to this point.

3.3.6 Exchanging data

The main loop of our implementation waits to receive a segment (using an `Offer` session type) and branches based on its type (section 3.3.1) and whether it is acceptable or not. Once it is processed the session type continues recursively and this repeats. Let us break down the `ServerSystemCommLoop` session type:

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

Acceptable with payload This is the "happy path" of data transfer.

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

First we acknowledge the data by sending an empty ACK. The data received in the ACK segment is passed to the server user in a message of type `Data`. The user can choose to send back a response in which case we send an ACK with a payload or initiate closing the connection in which case we send a FIN ACK. Since we send an acknowledgement either way, we could omit the first empty ACK but we keep it because we should send it immediately and not delay it until the server user continues.

Acceptable without payload Typically these are acknowledgements of data we have sent. In this case we simply pass the ACK to the TCP state machine to update the state and/or remove packets from the retransmission queue. We do not send anything in response and the session type reflects this:

```
Ack.ServerSystemCommLoop,
```

FIN ACK The peer has initiated closing the connection. In this case our TCP state machine will transition from ESTABLISHED to the CLOSE-WAIT state. We send back an empty acknowledgement in response and inform the user. Finally the continuation session type has a name and is defined separately, as it is recursive also.

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

Note that currently FIN ACKs that are not acceptable are not handled properly, as the branch for unacceptable segments is typed as `Ack`. To solve this would require another branch specifically for unacceptable FIN ACKs.

Unacceptable In this case we have received a segment whose sequence numbers are not acceptable and per the RFC we respond with an ACK which serves to inform the peer about our current receive window start and length.

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

Timeout The final case is a virtual message type to handle timeouts as described in section 3.3.7. In this case we retransmit the segment at the top of the retransmission queue:

```
Timeout.
    (RoleClientSystem + Ack /* retransmission */).
    ServerSystemCommLoop


  })
]);
```

This session type branches off to continuations named with the TCP state they correspond with: `ServerSystemCloseWait` and `ServerSystemFinWait1`. These are recursive themselves and serve to close the connection gracefully.

3.3.7 Retransmission

When a transmitted segment goes unacknowledged for a certain amount of time, we must retransmit it. Ordinarily this would be implemented using a retransmission timer and when it expires a segment from the top of the retransmission queue is sent asynchronously. The session type does not allow us to do this and we must specifically arrange for sending a retransmission ACK.

The simple session type theory we are using does not have a notion of timeouts but we can emulate them 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 continue recursively back to wait to receive another 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. This could also be used to implement a keep-alive mechanism and zero-window probing but we have not done either of these so far.

3.3.8 Closing the connection

Closing a TCP connection is a two-step process. Each direction of the stream is 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. See section 3.3.6 for the initial step. Now that the server has sent a FIN to the client, it must wait for its acknowledgement. Later, the client can still send more data, which we simply acknowledge and throw away, as we decided to not support a half-closed connection from the point of view of the server user, though in the future, the proper thing to do in this case would be to reset the connection instead. The client eventually decides to close too.

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

```
            end
    }
)];

Rec!(pub ServerSystemFinWait2, [
    (RoleClientSystem & {
        Ack. // data we don't care about
            (RoleClientSystem + Ack).
            ServerSystemFinWait2,
        FinAck. // other peer is closing as well
            (RoleClientSystem + Ack).
            end
    })
]);
```

In many situations the connection close handshake (initiated by the server) happens in two steps: first the server sends a FIN and then the client responds with an ACK. The client later decides to close as well and sends a FIN to the server. However, sometimes the client can respond with a FIN ACK immediately to the server's FIN. This is described by the second branch in the `ServerSystemFinWait1` session type above.

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 does not quite make sense under our call-and-response style of synchronous interaction – there is no opportunity for the server to decide to close while waiting for the client. Despite this there is a situation in which it can happen: if the server closes and the client closes too without waiting for a response to its previous message the timing can be such that the FIN-WAIT-1 session type above receives an ACK it is not expecting and things go wrong. This is a known limitation of our implementation, we believe there is nothing fundamentally preventing us from handling this situation.

Chapter 4: Evaluation

To evaluate our *server system* component (which was the focus of this project) 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. Having implemented both the *server system* and the *server user* we can now run the application and communicate with it over the TUN interface (section 3.2).

4.1 Linux TCP stack

We tested the TCP implementation primarily against the Linux kernel TCP stack, by running our program and connecting to it using a Linux user-space TCP client (netcat). The basic functionality of the server we tested is:

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.

In all cases we utilised a packet sniffer (Wireshark) to monitor the communication. We have made sure that the connection closes cleanly with no RSTs in all of the basic cases.

In addition to this we have used an interactive Python shell to connect a socket to the session typed server in order to test some other cases. One of these is that the server must be able to accept data even after it has sent a FIN to the client. It should also be possible for the server user to send data after the client has sent a FIN. We have verified that this works correctly but removed the test from the code as it causes the usual netcat test to generate RSTs because the kernel has nothing to do with the data after netcat has closed.

4.2 Manual testing using Scapy

Scapy [13] is a packet manipulation framework for Python. It allows us to construct arbitrary packets and send them over the network. It provides a Python shell which is a powerful tool for interacting with the network manually.

We have used Scapy 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. All of these 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. Handling of unexpected segments is deficient in the states leading to connection closure. This is only an implementation limitation and could be resolved by adding all the possible branches to the session type and implementing the corresponding logic. However, this does point to a limitation of the session type approach in general, it leads to an explosion of possibilities and is cumbersome to implement.

4.3 Linux TCP stack with `netem` impairment

Finally we have tested our implementation against the Linux kernel TCP stack with the addition of simulated network errors. We have used the `netem` module to introduce packet loss, delay and reordering. This is possible thanks to the fact that we use a separate TUN interface for our server and can attach the `netem` queue discipline to it.

The usual test procedure was to first establish a connection with `netem` disabled (because we do not implement retransmission in the three-way handshake) and then enable it. As for the client we have used both netcat for manual testing and a Python script. The script configures the `TCP_NODELAY` option on the socket and sends messages in a loop with a small delay between them. This is to ensure that the client sends a lot of small packets and we can observe the effect of packet loss and reordering. The received data was then compared to the expected output.

We found that the server is able to handle these errors and the connection can recover once the impairments are lifted. The server does not cache any segments in the receive window if they are not in order. This is bad for performance but it is not a correctness issue. This could be improved in the future. The server also correctly retransmits segments for which it has not received an acknowledgement, making it robust to packet loss in the ESTABLISHED state. As mentioned earlier, the three-way handshake is not robust to packet loss and the connection can get stuck if any packets are lost during the handshake. The same applies to the states after closing has been initiated by either side.

4.4 Limitations

Our implementation of the TCP protocol is not complete and does not implement all of the features of the protocol. Furthermore, it is not compliant with the requirements of the RFC [3]. In this section we discuss the limitations of our implementation and whether they are fundamental to the session type approach or simply a result of our implementation.

What we find in summary is that many features are impossible to implement using the simple and synchronous session type model we are utilising and a more powerful abstraction is required. Some other features are missing simply due to a lack of time to implement them or us deeming them not important for this demonstration.

4.4.1 Asynchronicity

In general TCP connections, messages can be sent asynchronously and with no relative ordering between the peers. Our session type model and implementation,

however, make assumptions about this and require a strict call-and-response message ordering. This is discussed in more detail in section 3.3.3. This is a fundamental limitation of the simple session type system we are using, though asynchronous session type theories are widely researched and the Less is more [12] formalisation also considers asynchronous session types. Implementing asynchronous session types and lifting this limitation would be an interesting path for future work as many of the limitations described below have similar underlying causes and could also be solved along the way.

4.4.2 Flow control and Congestion control

Implementing flow control turned out to be impractical in a synchronous context. In a standard TCP implementation, the sender would buffer outgoing data and only transmit it when it makes sense to do so based on the Sender's algorithm [3, § 3.8.6.2.1]. It is, however, difficult, if not impossible, to describe such a behaviour using a synchronous session type. The sender would require a selection with a branch to send a packet and a branch to send nothing. While that itself is easy to implement, the counterpart is impossible, as there is no way to determine if nothing has arrived because not enough time has passed or because the sender has decided to send nothing.

This is a problematic limitation as flow control and the sender's algorithm are crucial features of TCP. It is likely that this could be resolved using a more powerful session type foundation featuring buffers and asynchronous communications, though we have not explored this avenue.

Flow control also ties in with congestion control which, broadly, is a set of mechanisms controlling when data can or cannot be sent based on more sophisticated algorithms modelling the congestion of network links. Congestion control is also an important feature of TCP and the specification requires implementations to support it. However, it is not implemented in this project for the same reasons as flow control.

4.4.3 Messages instead of stream

The *server user* interacts with the system in terms of `Data` messages each containing a vector of bytes of arbitrary size. In a typical TCP implementation these messages would be taken from or appended to an internal buffer. Since we perform no buffering in the system any data received from the user is immediately transmitted. And, since there is no reliable way for a receiver to tell how many segments will arrive, we cannot segment the data. Instead, we transmit it in a single segment. However, there are size limitations for segments, given by the negotiated MSS value. Our implementation simply ignores these limitations and breakage will occur if the user attempts to send too much data at once.

One way to remedy this while still avoiding having to introduce buffering into the session type is to only transmit as much data as is allowed within the MSS and to provide feedback to the user about how many bytes were transmitted. The user could then keep track of their own output buffer.

4.4.4 Delayed ACKs and Nagle's algorithm

Both the delayed ACK algorithm and Nagle's algorithm lead to a situation where the implementation can decide to send a message or send nothing. As we have seen

before in section 4.4.2, this is troublesome to represent using our session type system. Absence of these features leads to performance issues if this implementation were to be used in practice.

4.4.5 Various implementation shortcomings

Here we will summarise issues which are implementable without running into fundamental limitations, but we have not implemented them in our prototype. This list is likely not exhaustive and this fact only highlights the difficulty of implementing TCP fully and correctly.

RTO determination. To determine the value of the Retransmission timeout (RTO) measurements of the RTT must be taken and a non-trivial algorithm must be used to compute the resulting RTO value. We have not implemented any of this and used a trivial exponential backoff with an upper limit. This is good enough for the demonstration, as we are not attempting to achieve good performance characteristics.

Simultaneous close. As described in section 3.3.8, there is a situation in which both peers attempt to close the connection simultaneously. In a production TCP stack this must be handled properly. We skip implementing this feature as it does not quite make sense in a synchronous context.

Robustness. We have made sure that in the main ESTABLISHED state, receiving unexpected packets is handled correctly and the implementation can recover from all kinds of network issues. The same meticulous handling should be employed in all states, this however requires a large amount of effort and we have not fully accomplished it.

Resetting. A TCP implementation should be able to reset the connection upon receiving an RST segment at any point in its lifecycle. This is not implemented and would lead to an explosion in the number of branches in the session type as resets can be received virtually at any point. A viable solution may be to handle them "outside" the session typed logic.

Performance. We have made no attempt to achieve high performance, both in terms of the implementation itself (i.e. how optimised the code is) and how well it could utilise network links in practice. This was not a goal in the first place, however. We note that if performance were an important goal, the implementation would be significantly more difficult.

Zero window probing. If the peer's receive window size is zero, the implementation is required to periodically probe whether or not it has re-opened, as the window update ACK could have gotten lost. This feature is not implemented, but the infrastructure to do it is present, see section 3.3.7.

Chapter 5: Conclusion

In this project we have modelled TCP of the Internet protocol suite [3] using Less is more multiparty session types [12] 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. The session types were encoded into native Rust types and instances of them used as tokens to prove that the owner of the token is allowed to perform an operation described by the session type token.

We have successfully modelled a subset of TCP and tested that it works correctly using manual testing against the Linux kernel TCP stack as well as manually constructed TCP segments. We have used multiparty session types, as we have modelled both the TCP/low-level interface and the user/TCP interfaces using session types. The user communicates with the TCP implementation using a session typed channel instead of method calls. The implemented TCP features include the three-way handshake for accepting connections, exchanging data segments, generating acknowledgements as necessary including when unexpected segments are received from the network due to misbehaving peers or network errors. In addition, the connection can be closed gracefully by either party (the user or the peer). Unexpected situations are handled in many but not all parts of the implementation.

However, the use of a synchronous session type calculus does not align well with the highly asynchronous and buffered communication presented in the TCP specification. This results in features such as flow control or delayed ACKs being impossible to implement. Furthermore, we were forced to make an assumption about the application protocol running on top of the TCP implementation and only consider protocols that exhibit a call-and-response communication pattern. See section 4.4 for a more detailed discussion of the missing features.

A positive observation we have made is that using Rust with session types in general is a very ergonomic workflow and the type and borrow checker are able to catch subtle mistakes, especially when recursive session types and loops are present. While defining multiparty session types manually is verbose and difficult to read, we have successfully implemented macros using Rust's `macro_rules!` infrastructure (section 3.1.1) which remedy this issue to a large degree by introducing a more concise syntax.

On the other hand, a major downside of this approach is a state explosion that would occur if we were to diligently model every single situation that can occur in the real world. This is caused partly by the fact that TCP was not designed to be modelled in this way and relies on many types of messages being sent at many different points in the implementation. But a larger issue is that we have to assume that messages can be lost in transit. This means that we can never be sure what type of message will arrive next and we have to handle virtually all types of messages in every offer operation of the session type.

Future work

We identify several avenues of future research in this area.

While there are areas in which the current implementation can be improved incrementally (section 4.4.5), these are not very important unless the fundamental limitations preventing a full TCP implementation are resolved.

Using a more powerful session type abstraction featuring asynchronous communication will alleviate several of the limitations we have encountered. We have discussed this in section 4.4.1 and section 3.3.3. This path leads to a more complete TCP implementation and possibly even one that is usable in real-world systems. A promising foundation could be the MAGπ language [2].

Our implementation of branching session types using nesting (section 3.1.2) is difficult to use, as the result of an offer operation is not a flat `enum` of all the possible branches but instead an unbalanced binary tree. To unwrap this structure an isomorphic tree of e.g. `match` statements is required. Instead, Rust's procedural macros can be used to generate the appropriate enum definitions and supporting code for branching types of arbitrary arity.

Appendix A: Source code

The source code associated with the project is attached electronically, archived from the git repository at commit `619ddf006`. The latest version will also be made available on GitHub at `https://github.com/sammko/tcpst2`.

The source code of the Rust crate is contained in the `src` directory. It consists of the following source files:

`main.rs`	Set-up logic and code for the two threads implementing the server user and system. Part of this file is described in detail in section 3.3.5.
`lib.rs`	Definitions of the roles and TCP session types.
`cb.rs`	Code for the channel between server user and server system roles, implemented using `crossbeam`.
`smol_lower.rs`	The low-level `smoltcp`-based implementation details of the of the network channel. See section 3.2.
`smol_channel.rs`	The network channel implementation including TCP message type definitions.
`st.rs`	Core scaffolding for session type definitions, such as the base traits and session types (`Action`, `OfferOne`, …)
`st_macros.rs`	Definitions of the helper macros we have developed for defining session types (`St!`, `Rec!`). See section 3.1.1
`tcp.rs`	contains the TCP state machine implementation and associated definitions. (section 3.3.2)

In addition, a shell script called `up.sh` is provided to create a TUN interface with the correct permission and set up an IP address on it. This script should be used before running the main project using `cargo run`.

Finally, we note that this implementation is based on work by Ivan Nikitin and contains some code from his initial prototype.

Bibliography

[1] Mark Allman et al. *Known TCP Implementation Problems*. RFC 2525. Mar. 1999. DOI: 10.17487/RFC2525. URL: https://www.rfc-editor.org/info/rfc2525.

[2] Matthew Alan Le Brun and Ornela Dardha. "MAGπ: Types for Failure-Prone Communication." In: *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*. Ed. by Thomas Wies. Vol. 13990. Lecture Notes in Computer Science. Springer, 2023, pp. 363–391. DOI: 10.1007/978-3-031-30044-8_14.

[3] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[4] Fernando Gont and Andrew Yourtchenko. *On the Implementation of the TCP Urgent Mechanism*. RFC 6093. Jan. 2011. DOI: 10.17487/RFC6093. URL: https://www.rfc-editor.org/info/rfc6093.

[5] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. "Language primitives and type discipline for structured communication-based programming." In: *Programming Languages and Systems*. Ed. by Chris Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. ISBN: 978-3-540-69722-0. DOI: 10.1007/BFb0053567.

[6] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types." In: *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 273–284. ISBN: 9781595936899. DOI: 10.1145/1328438.1328472.

[7] Raymond Hu, Nobuko Yoshida, and Kohei Honda. "Session-Based Distributed Programming in Java." In: *ECOOP 2008 – Object-Oriented Programming*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 516–541. ISBN: 978-3-540-70592-5. DOI: 10.1007/978-3-540-70592-5_22.

[8] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. "Session Types for Rust." In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. WGP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 13–22. ISBN: 9781450338103. DOI: 10.1145/2808098.2808100.

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

[10] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. "Multiparty Session C: Safe Parallel Programming with Message Optimisation." In: *Objects, Models, Components, Patterns*. Ed. by Carlo A. Furia and Sebastian Nanz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 202–218. ISBN: 978-3-642-30561-0. DOI: 10.1007/978-3-642-30561-0_15.

[11] Vern Paxson. "Automated packet trace analysis of TCP implementations." In: *Proceedings of the ACM SIGCOMM'97 conference on Applications, tech-*

nologies, architectures, and protocols for computer communication. 1997, pp. 167–179.

[12] Alceste Scalas and Nobuko Yoshida. "Less is More: Multiparty Session Types Revisited." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290343.

[13] Scapy community. *Scapy*. https://scapy.net/.

[14] whitequark et al. *smoltcp*. https://github.com/smoltcp-rs/smoltcp.