todo

todo todo todo
todo todo todo

***Abstract*—TODO:**
***Index Terms*—TODO:**

I. Introduction

II. Background And Motivation

A. Internet Protocol Standardisation

B. Session Types

Session types are a behavioural typing discipline that were first introduced by Honda [1] as a typed formalism for concurrency. Session types represent a series of actions that each have a type and a direction of data exchange and can be used to formalise and verify message-passing based programs. Importantly, session types dictate the protocols that the channels must use to communicate. Hence, we can use the theory of session types to statically enforce that a protocol adheres to its type. As we will see later, using session types we can also guarantee that this protocol will have some desirable behaviours such as safety or deadlock freedom. To express protocols where two or more participants interact, *multiparty session types* [2] (MPST) can be used.

It is easiest to understand MPST by looking at an example of a protocol:

$$\begin{aligned} \Gamma \;=\; & s[a] : b \,\&\, l1(int).c \oplus l3(int).end, \\ & s[b] : c \,\&\, l2(int).a \oplus l1(int), \\ & s[c] : a \,\&\, l3(int).b \oplus l2(int).end \end{aligned} \tag{1}$$

In this typing context we can see communication occur between three participants: **a**, **b** and **c**. Each participant has a local view of their communication chain, denoted by the **s[role]** syntax. The local view of each is created by chaining session types that make up the overall session type of that participant. We use $\oplus$ and $\&$ to denote select and offer respectively. Intuitively, selection is a sending action between one or more choices. Similarly, offer is a receiving action between one or more choices. We use $l_{i \in I}$ to denote labels and the type between brackets denotes the message type. Each session type is also annotated with the participant that the action is directed towards. So, **b**$\&$**l1(int)** denotes receiving a message from participant **b** on label **l1** of type **int**. The continuation of each session type is denoted by a full stop. The **end** session type indicates the completion of communication.

A more complex concept used in MPST theory is recursion:

$$\begin{aligned} \Gamma \;=\; & s[a] : \mu T.b \oplus \{l1(int).T, \\ & \qquad\qquad l2(str).end\}, \\ & s[b] : \mu T.a \,\&\, \{l1(int).T, \\ & \qquad\qquad l2(str).end\} \end{aligned} \tag{2}$$

The type variable **T** and least fixed point μT are used to model recursion through unfoldings. We write $\mu T.S$ to say that μ binds **T** in the session type **S**. In this protocol **a** can either keep sending an integer to **b** recursively or opt to send a string on label ***l2*** and end the communication. Other important session type theory extensions, from the perspective of network protocols, are *asynchrony* and *timeouts*. These will be discussed in later sections.

As previously mentioned, session types can also be used to guarantee desirable properties. To do so, MPST theory uses the notion of *safety invariants*. Type systems are parametric on the safety invariant, which can be instantiated depending on the properties that the protocol must adhere to. The safety invariants present in the theory of MPST used for this work are *safe*, *deadlock-free*, *terminating*, *never-terminating*, *live*, *live+* and *live++*.

Identify applicable funding agency here. If none, delete this.

Each invariant also has judgments that must hold for the typing context to exhibit the corresponding property. Intuitively, *safe* ensures that every input is matched by a corresponding output and vice-versa. This means that continuations are actually reachable via transitions. *Deadlock-free* ensures that communication can make progress unless it has been closed, that is, all typing contexts must reduce to *end*. *Terminating* and *never-terminating* check whether communication either ends or never ends. Finally, various levels of strictness of liveness ensure that all input and output actions eventually occur.

Looking at the typing context in 1, if we instantiate Γ with ϕ = *safe*, and derive the needed typing judgement then we will see that this protocol type checks. Hence, a protocol adhering to this session type can be expected to exhibit a similar property at runtime. However, by instantiating ϕ = *deadlock-free*, and deriving the typing judgement, we will see that the context is not well typed. Hence, we can expect that a protocol adhering to this session type will deadlock. To emphasise, we can make a *static* typing judgement to derive the *runtime* behaviour of a process that abides by Γ. This distinction is important as the typing context properties are *decidable*, while the runtime process properties are not.

$$
\begin{array}{rcl}
c & ::= & s[r] \colon S \\
S & ::= & r \oplus \{l_i(P_i).S_i\}_{i \in I} \mid r \,\&\, \{l_i(P_i).S_i\}_{i \in I} \\
 & & \mid \mu(t).(S) \mid t \mid end \\
P & ::= & \{TcpPayloadTypes\}
\end{array}
$$

Fig. 1: The syntax of session type context language used to create the TCP model in $mpstk$.

The flexibility of session type theory can also be seen in the tooling that was created to facilitate making typing judgements. Specifically, the *mpstk* [2] tool translates safety invariants to the first-order modal μ-calculus used by the *mCRL2* [3] formal specification language. This means that the associated model checking toolset can be used to make typing judgements. This is done by creating a model of a protocol in the typing context language used by *mpstk* and then simply specifying the invariants to check and running the tool. For this work, we have further modified *mpstk* to include the payload types needed for TCP.

The syntax for the session type language that the protocols are written in is presented in Figure 1. We do not omit this as the syntax of the language used for $mpstk$ differs to that of LM. Additionally, we only need to consider a simplified subset of the calculus. Note that P can not be an instance of S, that is, we do not consider sending sessions across channels. Although this is standard to the theory of π-calculus, TCP would not benefit from such a mechanism. Hence, the payload types are restricted to the defined TCP payload types.

III. Session Typed TCP - Model

A. Scope Definition

We have defined the scope of the model to be the main data communication state machine of the protocol, as described in the TCP RFC [4] in the state machine overview. Generally, our model expresses the initial call to open and create a TCB, connection establishment, the main data communication loop, retransmission and connection closing. We have opted to omit edge cases where no new modelling techniques would be exhibited as these are trivial to add analogously to other cases in the model. We can express all described cases of opening a connection, namely the basic three way handshake, simultaneous connection synchronization, recovery mechanism from old duplicate SYN, and half-open connections. We model TCP connection failure given a set R2 threshold and can model communication given that this is set to infinity. Finally, in terms of connection closing we are again able to express all described cases - the normal close sequence, the simultaneous close sequence and half-closed connections.

Given this definition of a TCP model there are important limitations that must be noted. Firstly, while the theoretical language presented by Scalas and Yoshida [2] does introduce asynchrony, the tooling does not. The translation given between MPSTs and the model checking calculus is synchronous. This omission means that all send and

receive actions occur as soon as they are called, which is in contrast to the buffered communication of TCP. Additionally, synchronous communication does not permit us to model any flush mechanisms, urgent data calls or aggregating calls on the buffer for features such as delayed ACKs. To reiterate, while the theory of MPST could model all of these on paper, there does not exist the tooling to support the theory.

Secondly, our chosen calculus does not have timeouts as a primitive. Hence, we have modelled some parts of TCP using implied timeouts. That is, we assume that a timeout is triggered by a part of the implementation not connected to the session typed state machine. Once this timeout fires the implementation would pass this as a message to the state machine and a transition will occur. In the theory of session types there do exist languages with timed features [5] and timeouts as a primitive [6]. Notably, the session typed calculus presented by Bocchi *et al*. [5] has the ability to model both time bound and delayed actions in protocols using binary session types. In terms of MPST theory, the $MAG\pi$ language [6] defines timeouts, however, it is unclear how timeout branches define timeouts and if each of these timeouts has to be the same. Additionally, $MAG\pi$ does not consider sending messages after a delay, a feature that would be useful to model TCP. Further, these languages do not give us the ability to modify timeouts at run-time, which would need to be handled outside the scope of session types. Similarly to asynchrony, there also does not exist any tooling nor implementations of such languages that could be used for this work.

B. Session Typed TCP

With the scope of the model defined, we now discuss how we expressed TCP using session types. As previously discussed, we opt to use multiparty session types to model the TCP protocol. Multiparty communication allows us to express TCP closer to its RFC description as opposed to the usual binary session type view of the protocol. While the binary session type view treats server and client as the only two participants, we explicitly model the communication between the system implementation of TCP and the userspace. Hence, we have extracted the following roles - the userspace of the server, the server side implementation of TCP, the client side implementation of TCP and the userspace of the client. The server side of the communication is described by the **server_user** and **server_system** participants and correspondingly the client side consists of the **client_user** and the **client_system**. These four roles capture the majority of the state machine described in the RFC, but it is possible to add more roles to explicitly express some TCP features, for example, the protocol's error reporting to the IP layer. While this would be more accurate, it would also make the model more complex and was deemed out of scope for the initial session typed view of the TCP state machine.

```
client_system&syn(SynSet).
   client_system<+>synack(SynAckSet).
      mu(t)(
         client_system&ack(AckSet).
         server_user<+>read_queue(Data).
         server_user&write_queue(Data).
         client_system<+>ack(AckSet).t
      )
```

Listing 1: Model of the basic three way handshake and the data loop. Retransmission and connection closing omitted.

In the initial OPEN call we assume that one side is the server and the other side is a client. Hence, we only model the case where the server side requests a passive-open and the client side requests an active-open. It is also possible to express more fine-grained cases, for example, having an explicit type for each parameter needed for the call and handling each case. Adding these is trivial if needed and is analogous to adding more match cases and does not change the flow of communication.

After the TCB is created, the connection establishment phase begins. We model this through the basic three-way handshake. Connection establishment using the basic three way handshake and the data communication loop from the view point of the **server_system**can be seen in Listing 1. It is also possible to express the other cases for connection establishment, such as simultaneous open. Again, this is trivial to add and is omitted for readability. We opt to explicitly type the connection

establishment packets as *SegSynAckSet*, *SegAckSet* and *SegSynSet* instead of just *Seg*. This allows us to demonstrate the assurances that session types can provide. By extracting these flag combinations as types, we enforce that during connection establishment we must exhibit the correct pattern. That is, the server must receive a segment with the SYN flag sent and respond with a segment with the SYN-ACK flags set. Hence, we do not need to explicitly model the client sending unwanted segments and the server then handling these. This is because a TCP implementation written using an implementation of session types would not be able to send segments with wrong flags set, assuming a well behaved parser. So, by being explicit in our types we have more control over the segments being sent which gives us flexibility in how we want to model the protocol. We can choose to be more general and handle error cases explicitly, or we can be more constrained in what we are allowed to send by having narrower types and disallow such edge-cases from occurring.

Once the handshake is complete we initiate the data communication loop by instantiating a recursive variable on each side. On the server side, the recursive block accepts a segment from the client and in the case of an accepted sequence number sends that data to its user. The server then reads from the user and sends a segment with data to the client.

```
server_user&close_init(Close).
client_system<+>fin(FinSet).
client_system&{
      // USER INITIATED CLOSE
      fin_ack(FinAckSet).
      client_system<+>ack(AckSet).
      server_user<+>close(Close).end,
      // SIMULTANEOUS CLOSE
      fin(FinSet).
      client_system<+>ack(AckSet).
      client_system&ack(AckSet).
      server_user<+>close(Close).end
   }
```

Listing 2: Closing the connection via user initiated close and an example of simultaneous close.

The user can also respond with a CLOSE call, starting the connection closing phase of the protocol. We model the cases where either server or client initiate the closing handshake by sending a segment with the FIN flag set. Note that due to the underlying language not having timeouts we do not explicitly model the TIME-AWAIT state. Instead this is an implied external transition, the server sends the user a CLOSE once the timeout fires outside the session typed state machine. Listing 2 shows a user initiated CLOSE call and a possible simultaneous close from the view point of the server.

```
rto_exceeded(AckSet).
client_system<+>retransmit(AckSet).
client_system&{
   ack(AckSet).t,
   retry_threshold_exceeded(RstSet).
   server_user<+>
       connection_aborted(Close).end
}
```

Listing 3: Model of retransmission of a lost segment.

During data communication it is also possible that we receive an unacceptable segment, in which case retransmission occurs. In the retransmission cases, the server either successfully receives an acknowledgement for the retransmitted segment or the retry threshold is exceeded and the connection is aborted. Retransmission initiated by the server can be seen in Listing 3. As previously mentioned, the timeout is implied. We have not put retransmission everywhere where it is possible as this would make the model unreadable, while not showcasing any other modelling techniques. It is technically possible for retransmission to occur at any stage of the communication after the very first segment is sent. Hence, we would need to copy the retransmission block after every communication. On the client side, data communication, connection closing and retransmission are analogous.

IV. Multiparty Implementation

V. Related Work

VI. Conclusion

VII. Acknowledgements

References

[1] K. Honda, "Types for dyadic interaction," in *CONCUR'93*. Springer, 1993, pp. 509–523.

[2] A. Scalas and N. Yoshida, “Less is more: Multiparty session types revisited,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019. [Online]. Available: https://doi.org/10.1145/3290343

[3] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse, “The mcrl2 toolset for analysing concurrent systems,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 21–39.

[4] W. Eddy, “Transmission Control Protocol (TCP),” RFC 9293, Aug. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9293

[5] L. Bocchi, M. Murgia, V. T. Vasconcelos, and N. Yoshida, “Asynchronous timed session types,” in *Programming Languages and Systems*. Cham: Springer International Publishing, 2019, pp. 583–610.

[6] M. A. L. Brun and O. Dardha, “Mag*p*: Types for failure-prone communication,” 2023.