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

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

safety invariants present in the theory of MPST used for this work are *safe*, *deadlock-free*, *terminating*, *never-terminating*, *live*, *live+* and *live++*. 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.

C. A Session Typed View Of TCP

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

Fig. 1: The syntax of session type contexts used to create the protocol models in $mpstk$.

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, the presented network protocols would not benefit from such a mechanism. Hence, the payload types are restricted to the defined TCP payload types.

D. Extracting the model

The breakdown of which TCP features we are able to model in our chosen MPST theory is given in Table **??**. Note that "Outside Of Scope" means that the specific part of the protocol does not pertain to communication. Many of these features involve changing state variables at runtime, performing dynamic assertions on parsed data, and taking actions that do not involve other participants. We may or may not be able to express these features as separate algorithms using session type theory or other formal mechanisms, but they will not be considered in this work as they do not directly involve the TCP state machine.

TODO: Discussion of what is modellable TODO: Discussion of what is not modellable and why

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. TODO: Check if this is the first ST model that considers TCP to be multiparty? 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.

Our model expresses the initial call to open and create a TCB, connection establishment, the main data communication loop, retransmission and connection closing. The matchings from the RFC to the corresponding parts of the model can be found in Table **??**. Note that for each RFC exert we do not show every place where we model this behaviour. Rather we show an example of it and the corresponding line number to show how this can be expressed.

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. 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 do, however, 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.

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. There are languages that use timeouts in session types [**?**] but currently there are no implementations nor tooling based on this theory that could be used for this work.

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. 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. This is further noted in our discussion on the usability of session types in Section **??**.

On the client side, data communication, connection closing and retransmission are analogous.

TODO: What have we not modelled that is important for TCP?
Possibly:

- Dynamically setting RTO
- Reset generation
- Sending reset (trivial to add but not an interesting case)
- Keep-alives
- Retransmission queue is not explicit since this is not part of the communication between roles (could be expressed as another role?)
- Communicating errors to IP layer - new role?
- Congestion control external state machine - could be yet another role?
- Zero-Window Probing - implicit in data communication, could be made more explicit via message types
- Delayed Acknowledgments - we dont have segmentation semantics, its implied that one message = one segment sent but we cant reason about "we sent X segments, now send an ACK", I think this can be done with context free STs?

III. Multiparty Implementation

IV. Related Work

V. Conclusion

VI. 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