tcpst todo

todo

February 23, 2023

1 Introduction

2 Background

2.1 Internet Protocol Standardisation

2.2 Session Types

Session types are a behavioural typing discipline that were first introduced by Honda [2] as a typed formalism for concurrency. Session types represent a series of actions that each have a type and a direction of data exchange. Importantly, session types dictate the protocols that the channels must use to communicate. Consider an example where a client **C** sends two strings to a server **S** representing the username and password of the user needed for authentication and expects a boolean, confirming if the user is to be authenticated and then terminates:

$$\mathbf{C} \;=\; \texttt{!String.!String.?Boolean.end}$$

This session type describes that **C**, sends (`!`) two strings and then receives (`?`) a boolean and then terminates (`end`). From the perspective of the server, we have the *dual* session type:

$$\mathbf{S} \;=\; \texttt{?String.?String.!Boolean.end}$$

We can see that the server cannot proceed in a way that would be incompatible with the client. This is guaranteed by the notion of *duality*, which means that the sending and receiving sides have consistent typing. For example, when **C** sends a string, **S** expects to receive a string.

We can further understand the importance of duality in a slightly more complex example. Consider now that client **C** can first choose an authentication method, credentials or token, provide the chosen method and terminate, while the server **S** will offer two authentication methods and then proceed accordingly depending on the method chosen by the client and terminate:

$$\begin{aligned}\mathbf{C} \;=\; \oplus\{\, & \mathit{cred} : \texttt{!String.!String.?Boolean.end},\\ & \mathit{token} : \texttt{!String.?Boolean.end}\,\}\end{aligned}$$

$$\begin{aligned}\mathbf{S} \;=\; \&\,\{\;& \mathit{cred} : \texttt{?String.?String.!Boolean.end},\\ & \mathit{token} : \texttt{?String.!Boolean.end}\;\}\end{aligned}$$

We now have a session type where the client chooses ($\oplus$) one of the possible branches that the server offers (&), which is represented via the constructs of *selection* and *branching*. Due to the duality concept, the client can't select an authentication method not provided by the server. Hence, we can see how duality ensures *communication safety* by ensuring the compatibility of session types.

Having defined what is meant by session types, it is important to note that the examples provided used *binary session types*; that is, all communication is handled on one channel by two participants. To model more complex protocols where two or more participants interact, *multiparty session types* [3] (MPST) can be used. In classic MPST theory, we define a *global type* that describes the communication of all participants and then *project* this into *local session types*, which represent the communication from the viewpoint of each participant. While the classic MPST theory gives us a global view of communication, it is also overly restrictive in the processes that can be typed due to the consistency restriction. In this work we use the more general multiparty session type theory (LM) presented by Scalas and Yoshida [7]. Unlike the classic theory, LM does not base itself on the concepts of global types and duality. This decoupling from binary session type theory makes the LM calculus more general.

The LM view of multiparty session types is based on the concept of a weak typing context *safety invariant*. A LM type system is parametric on the safety invariant, that is, by changing the parameter we can control what processes are typed, based on the properties that we want. The weakest of the invariants is simply safety. The safe invariant ensures that we have corresponding branches for selections, that recursive unfoldings are safe and that reductions are also safe if the context being reduced is safe. In order to be typeable under any LM type system a protocol must be at least safe. The authors present several typing context properties centered around deadlock freedom and liveness. An example of a protocol in LM looks like the following:

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

In this typing context we have a communication between three participants: `a`, `b` and `c`. Each participant has a local view of their communication chain, denoted by the $s[\mathit{role}]$ syntax. Similarly, each action is annotated with the participant that the action is directed towards. We use $l_{i\in I}$ to denote labels and the type between brackets denotes the message type. Unlike the previous binary session type example we use $\oplus$ to denote sending and & to denote receiving. This is a purely syntactical choice - sending is a case of selection and receiving is a case of branching where each only have one option.

If we instantiate Γ with $\phi = \mathit{safe}$, 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 $\phi = \mathit{deadlock\text{-}free}$, 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 have made a *static* typing judgement to derive the *runtime* behaviour of a process that abides by Γ. This distinction is crucial as the authors show that the typing context properties are *decidable*, while the runtime process properties are not. To perform these typing judgements the authors show a further advantage of the flexibility of their theory. Specifically, its ability to be integrated into various systems. The authors express ϕ as a formula in the μ-calculus and develop the *mpstk* tool that translates session typed contexts to the *mcrl2* [1] model checker. Hence, typing judgements over various safety invariants can be made automatically.

3 Formalising Internet Protocols Using Session Types

Based on this, we leverage session type theory to aid network protocol developers during both the draft and implementation stages of protocol design. We demonstrate how a multiparty session type system can be incorporated and used in a systems programming language to enforce protocol adherence. We have chosen to focus on TCP as a demonstration of the tooling chain because TODO: .

3.1 Multiparty Session Type Implementation

3.2 Property checking

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

Figure 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.

3.3 Extracting the model

Line Number	Model Snippet	RFC Extract

1, 16, 77, 46	`server_system<+>tcb_new(TcbInfo)`
	`server_user&tcb_new(TcbInfo)`
	`client_system<+>tcb_new(TcbInfo)`
	`client_user&tcb_new(TcbInfo)`
		"Create a new transmission control block (TCB) to hold connection state information..."

17, 3	`server_user<+>error(DiffservSecurity)` `server_system&error(DiffservSecurity)`	"Verify the security and Diffserv value requested are allowed for this user, if not, return "error: Diffserv value not allowed" or "error: security/compartment not allowed""
49, 80	`client_user<+>error(RemoteUnspecified)` `client_system&error(RemoteUnspecified)`	"If active and the remote socket is unspecified, return "error: remote socket unspecified""
47, 78	`client_user<+>error(ConnectionIllegal)` `client_system&error(ConnectionIllegal)`	"If the caller does not have access to the local socket specified, return "error: connection illegal for this process"."
48, 79	`client_user<+>error(Insufficient...)` `client_system&error(Insufficient...)`	"If there is no room to create a new connection, return "error: insufficient resources""
50, 19	`server_system<+>syn(SegSynSet)` `client_system&syn(SegSynSet)`	"...if active and the remote socket is specified, issue a SYN segment."
19, 50	`client_system<+>syn_ack(SegSynAckSet)` `server_system&syn_ack(SegSynAckSet)`	"...TCP Peer B sends a SYN and acknowledges the SYN it received from TCP Peer A..."
53, 21	`server_system<+>acceptable(SegAckSet)` `client_system&acceptable(SegAckSet)`	"...TCP Peer A responds with an empty segment containing an ACK for TCP Peer B's SYN..."

19, 50	`mu(t)(...)`	"Once the connection is established, data is communicated by the exchange of segments..."
34, 64	`client_system<+>rto_exceeded(SegAckSet)` `server_system&rto_exceeded(SegAckSet)`	"For any state if the retransmission timeout expires on a segment in the retransmission queue, send the segment at the front of the retransmission queue again..."
26, 61	`client_system<+>retry_thresh(SegRstSet)` `server_system&retry_thresh(SegRstSet)`	"When the number of transmissions of the same segment reaches a threshold R2 greater than R1, close the connection."
88, 69	`client_system<+>close_init(Close)` `server_system<+>fin(SegFinSet)`	"The user initiates by telling the TCP implementation to CLOSE the connection..."
8, 29	`server_system<+>close_init(Close)` `client_system<+>fin(SegFinSet)`	"The remote TCP endpoint initiates by sending a FIN control signal..."
31, 71	`client_system&fin(SegFinSet)` `server_system&fin(SegFinSet)`	"Both users CLOSE simultaneously"
71, 85	`client_user<+>close(Close).end` `client_system&close(Close)`	"When a connection is closed actively, it MUST linger in the TIME-WAIT state..."

Table 1: TODO:

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 1. 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 [**todo**] 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?

RFC Place	Feature	Modellable	Not Modellable	Outside Of Scope
Section-3.1	Header Format			✗
Section-3.2	Specific Options Definitions			✗
Section-3.3.1	Connection State Variables			✗
Section-3-3.2	State Machine Overview	✗		
Section-3.4-3	Segment Acceptability Tests			✗

Section-3.4.1	Initial Sequence Number			✗
Section-3.4.2	Quiet Time		✗	
Section-3.4.3	Quiet Time		✗	
Section-3.5-3	Simplest 3WHS	✗		
Section-3.5-7	Simultaneous initiation	✗		
Section-3.5-11	Recover from Old Duplicate SYN	✗		
Section-3.5.1-3	Half-Open Connection Discovery	✗		
Section-3.5.1-6	Active Side Causes Half-Open Connection Discovery	✗		
Section-3.5.1-8	Old Duplicate SYN Initiates a Reset on Two Passive Sockets	✗		
Section-3.5.2-4.1.1	Reset on non-existing connection	✗		
Section-3.5.2-4.2.1	Reset in any non-synchronized state	✗		
Section-3.5.2-4.3.1	Reset in any synchronized state	✗		
Section-3.5.3-1	Reset validation			✗
Section-3.5.3-2	Continuations based on reset validation	✗		
Section-3.5.3-3	Data carrying reset segment			✗
Section-3.6-3.1	User initiates close	✗		
Section-3.6-3.2	Remote initiates close	✗		
Section-3.6-3.3	Simultaneous close	✗		

Section-3.6.1-2	Half-duplex close sequence	✗		
Section-3.6.1-3	Linger in TIME-WAIT for 2xMSL		✗	
Section-3.6.1-3	Reopen directly from TIME-WAIT	✗		
Section-3.6.1-5	Reducing the TIME-WAIT State			✗
Section-3.7	Segmentation			✗
Section-3.7.1	Maximum Segment Size Option			✗
Section-3.7.2	Path MTU Discovery			✗
Section-3.7.3	Interfaces With Variable MTU Values			✗
Section-3.7.4	Nagle Algorithm			✗
Section-3.7.5	IPv6 Jumbograms			✗
Section-3.8.1	Retransmission Timeout			✗
Section-3.8.2	TCP Congestion Control			✗
Section-3.8.3-2.2	Pass negative advice to IP layer			✗
Section-3.8.3-2.3	Close connection when threshold exceeded	✗		
Section-3.8.3-2.5	Inform application of delivery problem	✗		
Section-3.8.4-1	Set state to idle		✗	
Section-3.8.4-2	Send Keep-alives	✗		
Section-3.8.4-3	Decide When To Send Keep-alives		✗	

Section-3.8.5	Urgent Information		✗	
Section-3.8.6	Managing The Window			✗
Section-3.8.6.1	Zero-Window Probing			✗
Section-3.8.6.2	SWS			✗
Section-3.8.6.3	Delayed Acks		✗	
Section-3.9.1.1-2	Passive Open	✗		
Section-3.9.1.1-5	Multiple Concurrent Connections			✗
Section-3.9.1.1-8	Verify diffserv or security/compartment			✗
Section-3.9.1.1-14	Active Open	✗		
Section-3.9.1.1-15	Multihomed host OPEN			✗
Section-3.9.1.1-17	Reject invalid remote IP address			✗
Section-3.9.1.2-2	SEND first	✗		
Section-3.9.1.2-3	PUSH flags			✗
Section-3.9.1.2-9	Urgent data		✗	
Section-3.9.1.2-10	Send to implied remote socket	✗		
Section-3.9.1.2-11	STATUS call	✗		
Section-3.9.1.2-15	Return buffer address along with coded response			✗
Section-3.9.1.3-2	Unauthorized error	✗		
Section-3.9.1.3-3	Queue several receives		✗	
Section-3.9.1.3-4	Pass received PSH bit	✗		
Section-3.9.1.3-5	Fill buffer			✗
Section-3.9.1.3-6	Urgent data		✗	
Section-3.9.1.4-2	Connection not opened error	✗		
Section-3.9.1.4-2	Not authorised error	✗		

Section-3.9.1.4-2	Outstanding SENDs are transmitted		✗	
Section-3.9.1.4-2	Closing TCP peer gives up		✗	
Section-3.9.1.4-4	Error when re-open attempt before TCP peer replies	✗		
Section-3.9.1.5	STATUS call	✗		
Section-3.9.1.6	ABORT call	✗		
Section-3.9.1.7	FLUSH call		✗	
Section-3.9.1.8	Report soft TCP errors	✗		
Section-3.9.1.8-5	Disable error reporting	✗		
Section-3.9.1.9	Set Differentiated Services Field			✗
Section-3.9.2	Set Differentiated Services Field			✗
Section-3.10.1-2.1	OPEN Call	✗		
Section-3.10.2-2.1	Connection Illegal Error	✗		
Section-3.10.2-4.1	Send a SYN segment	✗		
Section-3.10.2-4.1	Urgent bit		✗	
Section-3.10.2-4.1	Queue data for transmission		✗	
Section-3.10.2-10.1	Segmentize the buffer and send it with a piggybacked acknowledgement		✗	
Section-3.10.2-10.2	Urgent flag		✗	
Section-3.10.3-2.1	Connection illegal error	✗		
Section-3.10.3-2.2	Connection does not exist error	✗		

Section-3.10.3-10.1	Queue request if insufficient incoming segments are queued		✗
Section-3.10.3-10.1	Insufficient resources error	✗	
Section-3.10.3-10.2	Reassemble queued incoming segments		✗
Section-3.10.3-10.3	Notify the user of the presence of urgent data	✗	
Section-3.10.3-10.4	Notify when TCP endpoint takes responsibility for delivering data to the user	✗	
Section-3.10.3-12.1	Satisfied RECEIVEs		✗
Section-3.10.3-12.1	Error: connection closing	✗	
Section-3.10.4-2.1	Error: connection illegal for this process	✗	
Section-3.10.4-2.1	Error: connection does not exist	✗	
Section-3.10.4-4.1	Error: closing	✗	
Section-3.10.4-6.1	Error: closing	✗	
Section-3.10.4-8.1	Form a FIN segment and send it	✗	
Section-3.10.4-10.1	Queue until all preceding SENDs have been segmentized, then form a FIN		✗
Section-3.10.4-13.1	Error: connection closing	✗	
Section-3.10.4-13.1	An "ok" if a second FIN is not emitted	✗	

Section-3.10.4-15.1	Queue until all preceding SENDs segmentized		✗	
Section-3.10.4-19.1	Error: connection closing	✗		
Section-3.10.5-2.1	Error: connection illegal for this process	✗		
Section-3.10.5-2.2	Error: connection does not exist	✗		
Section-3.10.5-4.1	Outstanding RECEIVEs returned with "error: connection reset"	✗		
Section-3.10.5-6.1	Queued SENDs and RECEIVEs given "connection reset" notification		✗	
Section-3.10.5-16.1	Respond with "ok" and delete the TCB	✗		
Section-3.10.6-2.1	Error: connection does not exist	✗		
Section-3.10.6	Various status calls	✗		
Section-3.10.7.1-2.1	Discard data			✗
Section-3.10.7.2	LISTEN STATE calls	✗		
Section-3.10.7.3	Blind reset attack mitigation			✗
Section-3.10.7.3	Various acceptability tests			✗
Section-3.10.7.3	Don’t send data on SYN segments	✗		
Section-3.10.7.4	Acceptability tests			✗
Section-3.10.7.2	Aggregate ACK segments			✗

Section-3.10.8	If user timeout expires flush all queues	✗
Section-3.10.8	Requeue timeout segment on the retransmission queue	✗
Section-3.10.8	If the time-wait timeout expires on a connection, delete the TCB	✗

Table 2: TODO:

3.4 Discussion of how this works together

4 Discussion On Limitations Of The Theory

TODO:

- Generally unclear how we should describe "how much data is being sent" - is this the number of send calls or does the message need to be extended to express this?
- There are calculi that do timed and timeouts but these do not have the tooling
- Implementation limitations - heterogeneous channels (whats the overhead?), no variadic generics in Rust, will probably encounter other

5 Future Directions

TODO:

- automation
- other properties we may want to parametrise?
- model checker reporting - where is property violated?

- considering other protocols to exploit other theory of STs? - context free, multiplicities?

References

[1] Olav Bunte et al. "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. ISBN: 978-3-030-17465-1.

[2] Kohei Honda. "Types for dyadic interaction". In: *CONCUR'93*. Springer, 1993, pp. 509–523. ISBN: 978-3-540-47968-0. DOI: 10.1007/3-540-57208-2_35.

[3] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types". In: *Journal of the ACM* 63.1 (Mar. 2016), 9:1–9:67. ISSN: 0004-5411. DOI: 10.1145/2827695. URL: https://doi.org/10.1145/2827695 (visited on 11/28/2021).

[4] Kenneth L. McMillan and Lenore D. Zuck. "Compositional Testing of Internet Protocols". In: 2019. DOI: 10.1109/SecDev.2019.00031.

[5] Stephen McQuistin et al. "Investigating Automatic Code Generation for Network Packet Parsing". In: 2021. DOI: 10.23919/IFIPNetworking52078.2021.9472829.

[6] Stephen McQuistin et al. "Parsing Protocol Standards to Parse Standard Protocols". In: 2020. DOI: 10.1145/3404868.3406671. URL: https://doi.org/10.1145/3404868.3406671.

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