If Layering is useful, why not Sublayering?

Rathin Singha[†], Rishabh Iyer[*], Charles Liu[†], Caleb Terrill[†], Todd Millstein[†], Scott Shenker[*], George Varghese[†]
[†]UCLA [*]UC Berkeley

ABSTRACT

The Internet's success arose from classical layering: protocols like TCP and Ethernet can be independently understood, changed, debugged, verified, and offloaded to hardware using a clean service interface between layers. To accrue the same benefits at a finer grain, we suggest *sublayering*, i.e., layering *recursively* within each layer. We show that the data link and routing layers have natural sublayers. However, while TCP intuitively decomposes into sub-functions (connection management, reliable delivery, congestion control) common state variables like sequence numbers and window sizes entangle these functions, making sublayering difficult. We propose an alternate sublayered TCP with equivalent functionality which enables easily changing congestion control and connection management. We also argue that sublayering can help create robust and verified Internet protocol implementations akin to seL4 for Operating Systems. To this end, we describe early experiments with a verified sublayered implementation of a simple bit-stuffing protocol using Coq, and a verified monolithic implementation of a lightweight TCP using Dafny. We end with a set of challenges for sublayered protocols.

CCS CONCEPTS

• **Networks → Network design principles**.

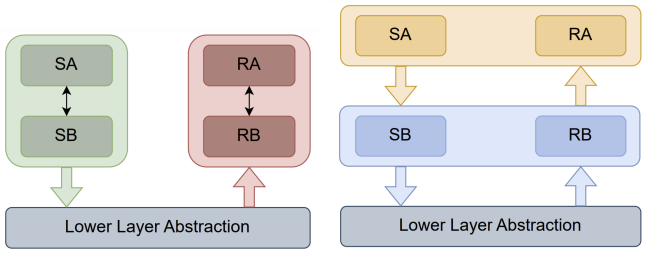
KEYWORDS

Sublayering, Modularity, TCP, Network Architecture

ACM Reference Format:
Rathin Singha, Rishabh Iyer, Charles Liu, Caleb Terrill, Todd Millstein, Scott Shenker, George Varghese . 2024. If Layering is useful, why not Sublayering?. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696348.3696892

HOTNETS '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696892



(a) Functional Modularity **(b) Sublayering Modularity**

Figure 1: Functional (left) vs. sublayered modularity (right).

1 LAYERING AND SUBLAYERING

> If anything is worth doing, it is worth doing to excess – *Edwin Land*

Layering—where each layer improves upon the services of the layer below to offer a superior service to the layer above [7]—is arguably the way all networks are built. Each layer contains functionality on different nodes that communicate with one another using packet headers, and this communication is sent through the API provided by the layer below. TCP, for instance, improves the datagram service of IP to reliable ordered byte streams using TCP headers with sequence numbers. In strict layering, each layer only looks at its header and interface data to do its job. While strict layering is often violated, it has enabled the engineering of the complex Internet and allowed it to accommodate vast changes in technology and applications over its 50 year history.

Despite its success, Internet layers have grown in complexity to add new functionality and improve performance, with each layer now comprising several intertwined distributed algorithms. For example, traditional transport protocols like TCP contain distinct components (e.g., connection management, reliability, congestion control, flow control) that are entangled through implementation and protocol design choices; newer transports like QUIC [20] further increase complexity by intertwining security and streams along with reliable delivery. This increasing complexity is not restricted to the transport layer, Other layers like the data link layer are also becoming more intricate, for example with the interposition of bridging.

Can we—in the same spirit as layering—conquer this complexity by breaking up a complex layer into *sublayers*? Taming complexity via sublayering differs from classical functional modularity. Figure 1 depicts a simple protocol like TCP with sender S and receiver R, that we sublayer into pieces SA, SB, RA, and RB, where SA peers with RA (Sublayer A) and

SB interacts with *RB* (Sublayer *B*). If the decomposition of *S* into *SA* and *SB* uses only functional decomposition, then this decomposition is purely internal to *S* and so is only useful in proving that *S* meets its specification. However, with sublayering, we have additional structure: *SA* interacts only with *RA*, and *SB* interacts only with *RB*, each acting as logical sub-protocols in a layered architecture. This structure allows us not only to modularize the reasoning about *S* but also the reasoning about the interactions between *S* and *R*.

Sublayering offers several advantages:

- **Fungibility and faster innovation:** The mechanisms used by an individual sublayer can be replaced, while maintaining the API and spec for that sublayer.
- **Debugging:** Like layering, sublayering has obvious pedagogic advantages in teaching, and allows complex layers to be taught in smaller, more manageable chunks. But it is also easier to understand and debug changes: we can localize bugs to sublayers (by examining which sublayer fails its contract) compared to a monolithic implementation.
- **Hardware offload:** Sublayering offers an alternative principle for hardware offloading —compared to the fast-slow path or functional modularity based decompositions used by earlier TCP offload engines like [16, 34].
- **Easier verification:** Sublayering offers a hierarchy that enables potentially simpler protocol verification compared to monolithic implementations because it modularizes reasoning about the distributed communication, thereby allowing different aspects to be reasoned about separately.

While we advocate for sublayering, we emphasize that *not everything should be a sublayer*, and so, we devise additional criteria to determine when a piece of functionality should be encapsulated into its own sublayer [7]. Subayering divides the functionality in a layer into separate pieces called sublayers subject to three tests:

T1. Sublayers are ordered such that each sublayer uses and improves the service of the sublayer below by adding a distinct function. Each sublayer does so by communicating with a peer sublayer in at least one other endpoint.

T2. Sublayers communicate with adjacent sublayers via a narrow interface.

T3. Each sublayer acts on separate packet bits, mechanisms, and states that are invisible to other sublayers, allowing sublayers to be independently replaced and verified.

This immediately begs the question as to how we differentiate layers and sublayers. We present additional principles to do so:

Public Interface: Layers maintain clear, well-defined public interfaces that the rest of the system depends on, while sublayers often do not.

Fine-grained services: Layers provide complete services to the upper layers while sublayers typically operate internally, providing fine-grained services within a single layer that the upper layers do not need to be aware of, adding flexibility and modularity to a layer without complicating the overall ecosystem.

Names: Layers often have names or identifiers, such as IP addresses for the IP layer, MAC addresses for Ethernet, or port numbers for transport protocols. By contrast, sublayers typically do not have such distinct identifiers but, instead, rely on the namespace of the layer.

As a simple example of applying these tests, buffer management in TCP should not be sublayered but be abstracted using functional modularity since it is a function local to a node. However, framing in the data link layer, which requires communication between a sender and receiver data link, while offering a distinct service (converting physical layer symbols to and from packets), is a good candidate. One can even break up sublayers recursively into further sublayers as we show for framing in Section 4.1.

This paper asks whether sublayering is useful, makes progress toward answering this question, and outlines the next steps. Thus the rest of the paper is organized as follows. Section 2 uses our tests to show the Data Link and Network layers are directly amenable to sublayering, but the transport layer (particularly TCP) requires some restructuring. Section 3 describes a proposed sublayering for TCP. Section 4 describes a vision for a verified Internet using sublayering; it includes verification experiments with a sublayered bit-stuffing protocol and a monolithic but lightweight TCP. Section 5 lays down challenges for verified sublayered implementations. Section 6 compares our approach to related work.

2 SUBLAYERING THE INTERNET

In this section, we first show how the data link (Section 2.1) and network (Section 2.2) layers are naturally amenable to sublayering. We then consider the challenge of sublayering the transport layer in Section 3.

2.1 Sublayering Data Link Layers

The Data Link layer can be divided into four sublayers: framing, error detection, error recovery, and encoding/decoding. (shown in Figure 2). Most Data Links from Ethernet to PPP begin by decoding the physical signals (encoded by the sender) into digital data followed by dividing them into frames so that headers can be determined as offsets in the frame; this makes encoding/decoding as the natural candidate for the lowest sublayer and framing should go on top of it. Error detection builds on framing by adding some form of checksum to the end of a frame to make the probability of undetected bit errors very small. In the case of reliable delivery like HDLC [4] and Fiberchannel [32], reliable delivery adds a header with

sequence numbers to guarantee delivery using retransmissions, but depends on error detection. Alternately, broadcast links like 802.11 dispense with error recovery and do Media Access Control (MAC) to guarantee that one sender at a time, eventually and fairly, gets access to the shared physical channel.

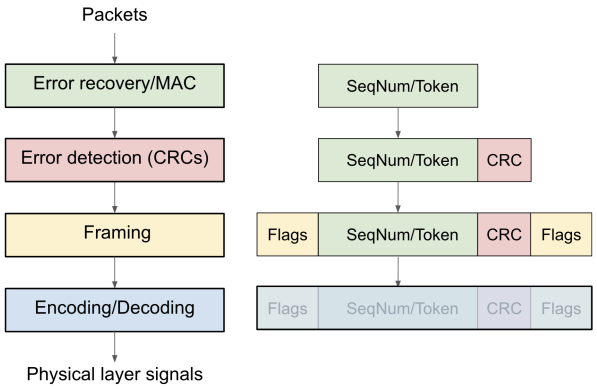


Figure 2: Data-link Sublayering

These sublayers meet the three tests we outlined earlier. For example, error detection improves the service of the framing sublayer by detecting errors with high probability, and has a simple interface to error recovery (frames with a flag indicating a bit error on reception). The details of how error detection is done can be confined to this sublayer, and the sublayer can be changed (to go from say CRC-32 to CRC-64) without changing other sublayers if the sublayers communicate only by their interfaces. The right side of Figure 2 shows how the header used by each sublayer can theoretically be added (and stripped on reception) – though actual implementations are unlikely to do this.

2.2 Sublayering Network Layers

Sublayering the Network Layer (Figure 3), is subtler as there is a clear separation between the data plane (forwarding) and control plane (routing). While this distinction is made explicit in Software Defined Networks [2], it is implicit in IP. Arrows denote information flow; solid lines denote the data plane and dashed lines the control plane. The path of a data packet passes directly from forwarding to the next hop Data Link. However, the forwarding database is itself built using routing, which we further sublayer into route computation and neighbor determination. Neighbor determination is the lowest sublayer because route computation needs a list of neighbors that is determined by handshake messages sent directly on the data link. Next, route computation is below forwarding because route computation builds the forwarding database.

The sublayers meet the three tests; test **T3** is met because the sublayers use completely different packets (e.g., LSPs versus IP packets), not merely different headers in the same packet. One can change say route computation from distance vector to Link State without changing forwarding. This separation is used in most implementations today where forwarding is offloaded to hardware.

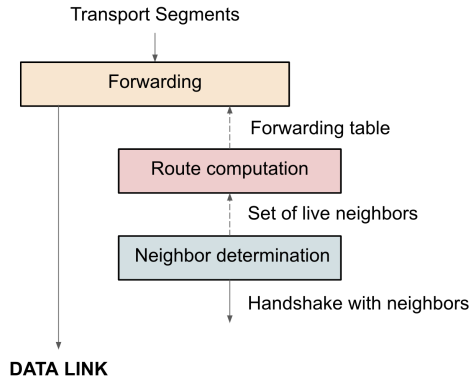


Figure 3: IP Sublayers

Figure 4: Network Sublayers: arrows represent information flow

2.3 Sublayering Transports?

Transports like TCP or QUIC have natural subfunctions (e.g., connection setup, reliable delivery, congestion control), and so, at first glance, it seems likely that they should also be amenable to sublayering. However, this is not the case since the state maintained by the transport layer (e.g., sequence numbers, window sizes, etc.) is shared by all of these subfunctions, which leads to non-modular code that is challenging to reason about. For example, even simple pseudocode examples of TCP, such as one on page 948 of the book TCP Illustrated Vol 2 [33] (not shown here), intersperse calls to several different functions such as demultiplexing (finding the right PCB block), connection management (checks for SYN_SENT, etc), reliable delivery (triggering fast recovery upon receiving duplicate acks), and congestion control (updating the window); all of which share and mutate the same state (encapsulated in the PCB block). Sharing state provides efficiency (the state is encapsulated into a memory-efficient layout) and improves performance (unrestricted access to the shared state avoids marshaling/unmarshaling overheads). However, it makes reasoning challenging, because reasoning about the correctness of a single function now requires reasoning about its interactions with all other functions via operations on the shared state.

So, we ask: *can we re-architect TCP to make it sublayered (and thus easier to reason about), while ensuring it continues to interoperate with existing TCP implementations?* We now outline a proposed sub-layering of TCP based on our three litmus tests for sublayers.

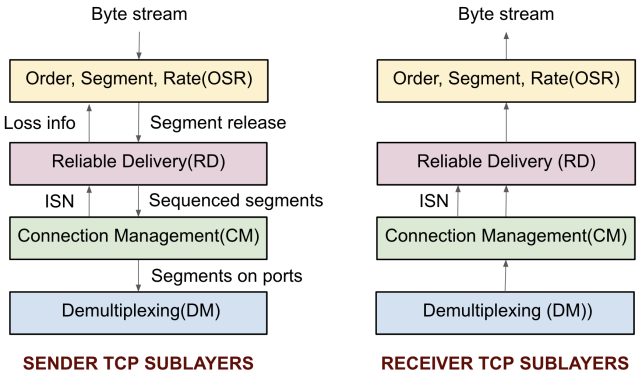


Figure 5: TCP Sublayers: arrows represent information flow

3 TOWARDS A SUBLAYERED TCP

Our proposed TCP sublayers are shown in Figure 5. The lowest demultiplexing (DM) sublayer is essentially UDP; it allows demultiplexing via standard destination and source port numbers. No sublayer can do its work without DM; so we place DM at the bottom. DM encapsulates details of binding IP addresses to ports and reusing ports. To pass test **T3**, DM only uses the destination and source port numbers as shown in Figure 6. The header as shown bears no resemblance to the standard TCP header in order to clearly separate sublayers. We will argue later that it is isomorphic (and can easily be translated) to the TCP header.

The next sublayer is connection management (CM) which encapsulates the classical TCP connection setup using SYN messages and disconnection using FIN messages. The main service it provides is to establish a pair of Initial Sequence Numbers (ISNs). The first standard specifying modern TCP (RFC793 from 1981) suggested choosing the initial sequence number to be unique in time using the low-order bits of a clock "to prevent segments from one incarnation of a connection from being used while the same sequence numbers may still be present in the network from an earlier incarnation" [28]. Faced with possible attacks against TCP sequence numbers, RFC1948 then proposed using a cryptographic hash of ports, addresses, and a secret key to decide the initial sequence number. This makes it hard for an attacker to predict the ISN. Regardless of the mechanism encapsulated, the main function of CM is to choose ISNs that are unique and hard to predict. Intuitively, CM sets up RD by providing a range of sequence numbers not present in the network so that segments and acks can be trusted as not being delayed duplicates (see Smith [29], p.145, for a formalization). We place the ISNs and the SYN/FIN flags in the CM portion of Figure 6.

The next sublayer is reliable delivery (RD), which uses the ISNs supplied by the lower connection management layer to reliably (i.e., exactly once) deliver segments given by the upper layer (OSR). OSR gives RD a segment identified by its byte offset, and RD translates this to segment sequence numbers (by adding the ISN). RD uses retransmissions to ensure the segment will eventually reach the receiver. All details of retransmission, including keeping track of a window of outstanding packets are encapsulated in RD; if Selective Acknowledgement is used, the SACK options are also processed by this sublayer though they are omitted in Figure 6 for simplicity, which only shows the normal sequence numbers in the RD header in addition to the ISNs in the CM header.

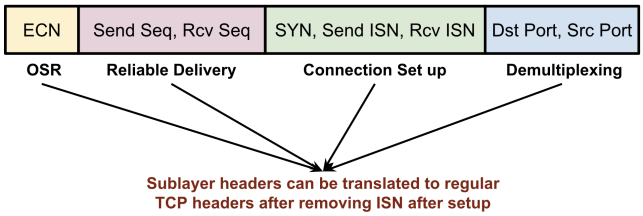


Figure 6: Redesigned TCP header

Finally, the uppermost sublayer provides Ordering, Segmenting, and Rate Control (OSR). OSR takes the byte stream and breaks it up into segments based on parameters like maximum segment size. At the receive end, segments may be delivered out of order by the RD sublayer. OSR must paste segments back in order (as in a standard TCP); the details of this reordering are hidden in OSR. OSR guarantees the main property of TCP—that the byte stream received is the same as the sent byte stream—using the properties that RD provides.

Finally, rate control is hidden within OSR which interfaces with the RD sublayer below by deciding when a segment is "ready" to be transmitted. Intuitively, the property it guarantees is if the network or receiver bottleneck rate changes and stays steady, the sending OSR will eventually reach and stay at that bottleneck rate. More elaborate performance properties can also be used [1]. Rate Control could potentially be separated from Ordering and Segmenting but this would require Ordering to pass a queue of segments to Rate Control (in addition to the byte queue at the client interface).

To implement congestion control in OSR, all congestion control signals should be available to OSR via either the OSR header or by the interface to RD (test **T3**). Thus explicit congestion control notifications like ECN are in the OSR subheader (Figure 6), as is the receiver window for flow control. Other congestion signals such as timeouts and loss information should be summarized and passed by RD to OSR as in [26]. Finally, the sending RD must tell the sending OSR when segments are acked so the sending OSR can advance the congestion and flow control windows.

In an alternative sublayering, congestion/Rate control can be thought of as providing a service [26] to reliable delivery (the window size), and hence arguably would be ordered *below* RD and above CM. But then the ISNs have to pass

through rate control. Instead, we found it cleaner to think of Rate Control as giving RD a segment when it is "ready" in order to meet a rate/flow control objective, placing rate control *above* RD.

We make the following case that this sublayering meets the three tests outlined earlier. **T1:** The sublayers are ordered and can be thought of as communicating with peer sublayers at the receiver. DM adds demultiplexing, connection management creates ISNs, RD does reliable delivery, and OSR converts the byte stream to and from segments, and adjusts the sending rate to network bottlenecks (congestion control) and receiver buffers (flow control). **T2:** OSR communicates with RD by deciding when to release a segment, RD communicates with CM by obtaining an ISN, and CM communicates with DM by specifying the 5-tuple for the connection. **T3**: This can be seen by inspecting Figure 6. If each sublayer adheres to its API, one could in principle seamlessly replace congestion control (by say a rate-based protocol) or connection management (by a timer-based scheme [31]).

3.1 Potential Objections

We address four possible objections:

Sublayering replicates functionality: Besides RD, CM does reliable delivery – but only for SYN and FIN. But this is implicit in TCP which uses a bootstrap reliability mechanism (retransmission and timeout of SYNs and FINs, no windows) to set up more sophisticated mechanisms in RD (windows, SACK, etc). Next, both RD and OSR need a "window"; for RD a window is the range of outstanding segments; for OSR it reflects a way to control the sending rate. These two concepts are conflated in TCP; it is reasonable to separate them. Similarly, both RD and OSR need mechanisms to advance their window when segments come out of order and a receiving segment fills a hole. This duplication of processing can be reduced if RD passes hints to OSR.

Sublayering does not help hardware offload: On the contrary, Figure 5 offers a principled way to offload parts of TCP processing to hardware. For example, OSR, which appears complex and likely to evolve, is best relegated to software. A simple decomposition places RD, CM, and DM in hardware; with more finagling and a modest duplication of state, only RD can be placed in hardware.

Sublayered TCPs cannot interoperate with standard TCPs: The header in Figure 6 clearly differs from the standard (RFC 793), making it impossible for a sublayered TCP – as stated so far – to interoperate with a standard TCP. However, we claim that the two headers are isomorphic. Our intent is that all information in the standard TCP header appear in Figure 6 and vice versa. While the ISN header is redundant, it is static after the initial handshake. Thus *adding a shim sublayer* that converts the sublayered header in Figure 6 to a standard TCP header, together with replicating all existing TCP functionality in some sublayer, should allow interoperability.

Sublayered TCP performance will be poor: Most performance issues in networking are due to protection, control overhead, and copying. We have already learned to finesse those for layer crossings, so why not for sublayer crossings? We return to this in the final section on challenges (Section 5).

We believe the potential disadvantages of a sublayered TCP are outweighed by potential advantages in faster innovation and debugging, cleaner hardware offload, and easier verifiability.

4 SUBLAYERING AND VERIFICATION

We now discuss whether sublayering can help construct verified implementations of Internet protocols. In the last 15 years, verified implementations of complex system software have come of age. For instance, seL4 [18] is a verified OS Kernel used in embedded systems with performance comparable to existing kernels; CompCert [23] is a verified compiler used in mission-critical software; Ironclad/Ironfleet [14] verifies practical distributed protocols; and NetCore [13] describes a machine-verified network controller for SDNs.

In contrast, while some initial efforts have been launched to build verified Internet protocol implementations of say TCP [3], QUIC [6] and IP routing [8], none seems either reasonably complete or had much uptake. Further, bugs in such software continue to this day [24, 37].

We describe two preliminary experiments to investigate whether sublayering can help verify Internet implementations. Section 4.1 describes our experience building a verified sublayered implementation of a simple but widely used protocol, bit stuffing, using Coq [15]. By contrast, Section 4.2 describes the difficulties of even verifying a straightforward property (in-order delivery) of a monolithic but simple implementation of TCP (lwIP TCP) using Dafny [22].

4.1 Verified Bit Stuffing

A commonly used framing mechanism within the Data Link is HDLC [4], which uses the bit pattern 01111110, called a flag, to delimit the start and end of a frame. To prevent unwanted flags in the data, HDLC uses a "stuffing" rule: whenever it sees 11111 in the data it adds (stuffs) a 0. After removing flags, the receiver "unstuffs" by removing a 0 after it sees 11111. Bit-stuffing appears simple, but subtleties make certain bit-stuffing rules fail. Some rules can have the stuffed bit form a flag with subsequent data bits; some flags can cause a false flag to occur using the data and a prefix of the end flag.

Sublayered Implementation: The standard implementation has the sender, in a single pass, emit a start flag, stuff the data on the fly, and finally emit an end flag. In a single pass,

the receiver looks for the start flag and unstuffs the data that follows on the fly, while also looking for an end flag which it strips. Instead, we suggest the following sublayering: the upper sublayer is a stuffing sublayer that does stuffing (at the sender) and unstuffing (at the receiver). The lower sublayer adds flags (at the sender) and removes flags (at the receiver). This is a *nested sublayering within framing*, which is itself a sublayer of the Data Link.

The sublayering meets our three tests– **T1:** the stuffing layer adds and removes the stuff bit, while the flag sublayer frames the data using flags; **T2:** the stuffing sublayer passes a frame without flags to and from the flag sublayer; **T3:** This holds as long as the flag is passed in the interface; a change in the interface (i.e., flag) would require a change in the stuffing rule. Thus, the correctness of stuffing depends on the flag: this shows up in the lemmas we proved,

Verified Implementation: We created a Coq implementation with functions for stuffing, adding flags, removing flags and unstuffing. The main specification is *Unstuff (RemoveFlags (AddFlags (Stuff(D)))) = D* for all data *D*. Our proof had 57 lemmas and 1800 lines of code. We also created a library of stuffing protocols that our proof deems valid; it found 66 alternate stuffing rules, some of which had less overhead than HDLC. Finally, we generated verified OCaml code from our Coq implementation.

Lessons learned:

1. Sublayering Modularity: The proof uses separate independent correctness lemmas for each sublayer which allows us to modularly reason about the distributed protocol.

2. Better stuffing rules: The flag 00000010 and the stuffing rule that stuffs a 1 after seeing the string 0000001 has an overhead (using a random model) of 1 in 128 compared to 1 in 32 for the HDLC rule. Verification can provide principled replacements for ad hoc constants that litter protocols.

4.2 Verifying a Simple TCP

We report our experience verifying a monolithic implementation of TCP which suggests the complexity of the task, despite two simplifications: we used a simple TCP implementation, the lwIP TCP stack [9]; we also only verified a simple in-order, reliable delivery property assuming the network is initially empty as an abstraction of CM's guarantee.

Dafny implementation: Our Dafny implementation follows the lwIP code which in turn is based on the BSD TCP code [33]. We wrote 30 lemmas and ≈ 3500 lines of code.

Lessons learned:

1. Partitioning large functions: Dafny times out for large functions like *tcp_input*. While *tcp_input()* breaks up into *tcp_process()* and *tcp_receive()*, *tcp_receive()* was also too large, and had to be broken up into smaller functions in ad hoc fashion. We did so based on cases such as: the window being updated on receiving an ack, a duplicate ack arriving, etc. In each case, we had to add many complex annotations to ensure that the postconditions of these ad hoc smaller functions were consistent with each other and *tcp_receive()*.

2. Ownership: Verification of monolithic stacks with unrestricted shared state (e.g., the PCB) is challenging because Dafny does not have an in-built notion of ownership [21]. Modifying the heap requires a plethora of annotations to manually specify the precise portions of the heap that an individual function accesses, to prove that functions do not interfere with one another via side effects in shared state.

3. Entangled State: For example, the window is crucial for ensuring reliable delivery, but reasoning is complicated because congestion/flow control can also alter the window.

Some issues are particular to Dafny and our proof strategy. Nevertheless, we conjecture that sublayering can mitigate them. Sublayering breaks up layer modules in principled, not ad hoc ways, and the state is segregated within sublayers. Further, once a sublayer is proved, we can forget the details of a sublayer, relying thereafter only on the postconditions of the lower layer. By contrast, many functions in our Dafny proof (that correspond to sublayers) require guarantees from the network; while this is a small cost, it suggests why $O(N^2)$ interactions may arise. Minimally, a sublayering exercise can suggest better proof strategies for monolithic code.

5 CHALLENGES FOR SUBLAYERING

We propose the following challenges for sublayered protocols that form a research agenda.

1. Refactor: Refactor monolithic implementations to be sublayered with APIs and a possibly re-architected header and test for basic functionality (e.g., reliable delivery for TCP) with a sublayered implementation at all nodes.

2. Interoperate: Show that the refactored implementation can interoperate with a standard (monolithic) implementation, possibly adding a shim layer to translate from the sublayered header to the standard header.

3. Tune: Use standard tricks to make the sublayered implementation perform close to the best monolithic one.

4. Verify: Verify, at least partially, the correctness of the implementation using Proof Assistants like Dafny or Coq, both with and without sublayering, to ascertain whether verifying a sublayered implementation is easier.

5. Replace: Replace some sublayers with alternatives and investigate the difficulty of doing so.

6. Hardware assist: Move some sublayers to hardware in an FPGA implementation to show that hardware assist can be done in a principled fashion.

While we plan to do so for TCP, the same research agenda holds for all protocols. Most protocol implementations today are not sublayered, including Data Links and network layers

despite the fact that their sublayers are natural and do not require extensive restructuring like TCP. Of particular interest to us is QUIC which has a clean sub-layering between networking (the transport layer) and security (the record layer). The transport layer can likely be further sublayered into a stream layer and a connection layer.

6 RELATED WORK

The closest work to ours is that of Ford et. al [11], who advocate breaking up the transport layer into three distinct layers—demultiplexing, congestion control, and reliable data delivery—to allow for flexibility and easier adoption of new protocol functionality. There are two key differences in our work: First, we focus on an existing protocol (specifically TCP), with the goal of breaking up its complexity for ease of verification. This focus allows us to define layers at a finer granularity, unlike Ford et. al whose focus on generality leads to a design that represents the lowest common denominator across different transport protocols. Second, we advocate for *sublayers* as opposed to layers. Doing so ensures that we do not modify the format of packet headers, which is critical for deployability in the public Internet [27].

Other related work includes SST [10] and Minion [27], who propose separating out ordered delivery from TCP and moving it into a separate layer. By doing so, SST and Minion allow applications to define their own policies for ordering, and potentially avoid the head-of-line (HOL) blocking inherent to traditional TCP. We see this work as a specific use case for sublayering, i.e., they seek to answer the question: "How do I sublayer TCP to avoid HOL blocking?" In contrast, our approach to sublayering TCP is more general, and subsumes this goal along with others (such as defining sublayers for hardware offload and verification).

Prior research on hardware offloading has primarily exploited functional modularity (as opposed to sublayers). For example, AccelTCP [25] proposes offloading connection management to NICs while the host handles reliable delivery. Similarly, TAS [17] proposes hardware offload for the fast path by bypassing the OS kernel for RPCs and a slow path for connection management and congestion control. Similar to us, all such work seeks to decompose TCP, but they do not use sublayers as we do to limit the interaction between different components.

Day's Recursive Internet Network Architecture (RINA) [12] has a superficial resonance with sublayering: both use recursion but in different ways. The primary goal of RINA is to expose similarity at all layers and reuse mechanisms at each layer. RINA does not, for instance, advocate sublayering TCP and Data Links as we do. Zave and Rexford's Real Network Architecture [36] finds new abstract models to fit Internet realities such as underlays and overlays. Kohler's Prolac TCP [19] uses standard functional modularity to make it more readable and modular. Narayan et al. [26] separate out TCP congestion control into a separate control plane with a well-defined interface in the same spirit as sublayering.

Existing work in verifying transport protocols [3, 5, 6, 8, 30, 35] automatically verifies message-oriented and not bytestream-oriented transports. Smith [29] verifies a *model* of CM and RD together (without sublayering) using manual verification in a proof of over 100 pages.

7 CONCLUSIONS

In a world of Internet protocols where complexity reigns due to the constant drumbeat of performance and functionality, how can the Internet's essential simplicity be restored? In this paper, we suggest sublayering as one way forward.

Sublayering has variants that differ from classical layering, yet meet the three tests. For example, control sublayers in the network layer (Figure 3) provide information for the data plane that bypasses them. Our sublayered TCP (Figure 5) has CM initially active and then silent.

This paper also proposes a second vision for building verified Internet protocol implementations. The NetCore paper [13] advocates using SDNs because "they relocate control from distributed algorithms running on individual devices to a single program running on the controller". We hope sublayering can remove some of the complexity of reasoning about distributed protocols.

REFERENCES

[1] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward formally verifying congestion control behavior. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21). Association for Computing Machinery, New York, NY, USA, 1–16.

[2] Martín Casado, Nick McKeown, and Scott Shenker. 2019. From ethane to SDN and beyond. SIGCOMM Comput. Commun. Rev. 49, 5 (nov 2019), 92–95.

[3] Guillaume Cluzel, Kyriakos Georgiou, Yannick Moy, and Clément Zeller. 2021. Layered Formal Verification of a TCP Stack. In IEEE Secure Development Conference, SecDev 2021, Atlanta, GA, USA, October 18-20, 2021.

[4] GL communications. 2024. HDLC Protocol Overview. https://www.gl.com/Presentations/HDLC-Protocol-Overview-Presentation.pdf.

[5] Andre Danthine and Joseph Bremer. 1978. Modelling and verification of end-to-end transport protocols. Computer Networks (1976) 2, 4-5 (1978), 381–395.

[6] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. 2021. A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer. In 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021.

[7] Edsger W. Dijkstra. 1968. The structure of the "THE"-multiprogramming system. Commun. ACM 11, 5 (may 1968), 341–346. https://doi.org/10.1145/363095.363143

[8] Mihai Dobrescu and Katerina Argyraki. 2015. Software dataplane verification. Commun. ACM 58, 11 (2015), 113–121.

[9] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science 2, 77 (2001).

[10] Bryan Ford. 2007. Structured streams: a new transport abstraction. In Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications. Kyoto, Japan, 361–372.

[11] Bryan Ford and Janardhan R Iyengar. 2008. Breaking Up the Transport Logjam.. In HotNets. 85–90.

[12] Eduard Grasa, Eleni Trouva, Patrick Phelan, Miguel Ponce de Leon, John Day, Ibrahim Matta, Lubomir T Chitkushev, and Steve Bunch. 2011. Design principles of the recursive internetwork architecture (RINA).

[13] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-verified network controllers. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 483–494.

[14] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In Proceedings of the 25th Symposium on Operating Systems Principles. 1–17.

[15] INRIA. 2024. The Coq Proof Assistant. https://coq.inria.fr/.

[16] Hankook Jang, Sang-Hwa Chung, Dong Kyue Kim, and Yun-Sung Lee. 2011. An Efficient Architecture for a TCP Offload Engine Based on Hardware/Software Co-design. J. Inf. Sci. Eng. 27, 2 (2011), 493–509.

[17] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP acceleration as an OS service. In Proceedings of the Fourteenth EuroSys Conference 2019. 1–16.

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 207–220.

[19] Eddie Kohler, M Frans Kaashoek, and David R Montgomery. 1999. A readable TCP in the Prolac protocol language. In Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication. 3–13.

[20] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In Proceedings of the conference of the ACM special interest group on data communication. 183–196.

[21] The Rust Programming Language. 2024. Understanding Ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[22] Rustan Leino and et al. 2024. The Dafny Programming and Verification Language. https://dafny.org/.

[23] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress.

[24] Jonathan Looney. 2019. CVE-2019-11478: TCP retransmission queue implementation in the Linux kernel can be fragmented when handling certain TCP Selective Acknowledgment (SACK) sequences. https://nvd.nist.gov/vuln/detail/CVE-2019-11478.

[25] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. {AccelTCP}: Accelerating network applications with stateful {TCP} offloading. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). 77–92.

[26] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18). Association for Computing Machinery, New York, NY, USA, 30–43.

[27] Michael F Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin, and Bryan Ford. 2012. Fitting Square Pegs Through Round Pipes: Unordered Delivery {Wire-Compatible} with {TCP} and {TLS}. In 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). 383–398.

[28] J Postel. 1981. RFC 793. https://www.ietf.org/rfc/rfc793.txt.

[29] Mark Anthony Shawn Smith. 1997. Formal verification of TCP and T/TCP. Ph. D. Dissertation. Massachusetts Institute of Technology.

[30] Carl A Sunshine and Yogen K Datal. 1978. Connection management in transport protocols. Computer Networks (1976) 2, 6 (1978), 454–473.

[31] Richard W Watson. 1981. Timer-based mechanisms in reliable transport protocol connection management. Computer Networks (1976) 5, 1 (1981), 47–56.

[32] Tom Weimer. 2003. Fibre channel fundamentals. Feb 25 (2003), 2.

[33] Gary R Wright and W Richard Stevens. 1995. TCP/IP illustrated, volume 2: The implementation. Addison-Wesley Professional.

[34] Zhong-Zhen Wu and Han-Chiang Chen. 2006. Design and implementation of tcp/ip offload engine system over gigabit ethernet. In Proceedings of 15th International Conference on Computer Communications and Networks. IEEE, 245–250.

[35] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. 2019. Verifying Software Network Functions With No Verification Expertise. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019.

[36] Pamela Zave and Jennifer Rexford. 2024. The Real Internet Architecture: Past, Present, and Future Evolution. Princeton University Press.

[37] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021.