Statement on the Revision of ⟨Eurosys 2021⟩ for:
ALCC: Migrating Congestion Control to the Application Layer in Cellular Networks
Based on the Reviewers’ comments

May 1, 2021

Prior to our submission of this paper to Jsys, we have submitted the paper to Eurosys 2021. We want to thank the reviewers of Eurosys 2021 for their feedback on the paper. We had a total of 13 reviews: 6 reviews from the main PC, and 7 reviews from the shadow PC. Out of the total 13 reviews, 10 of them gave the paper weak-accept, while the rest of the 3 gave the paper weak-reject. The weak-rejections were without convincing arguments on why the paper should be rejected apart from not making it clear how different is ALCC from the CCP framework. We want to make a few points clear. First, We are not the CCP authors. Second, the paper was not reviewed by CCP authors since we have marked them as conflict of interest given that we have a joint research grant with them. We believe that the paper was unjustly rejected because the reviewers had no insights into how the CCP works and did not quite understand the difference between ALCC and the CCP. Also, having 4 out of 6 reviewers accept the paper with only 2 weak rejects and having the paper shadow accepted only proves our point that Eurosys should not have rejected the paper.

We have reached out to the CCP authors and discussed ALCC; this was their reply on the main difference between CCP and ALCC: “At a high level, I think the paper could do a better job of making the points you make in the rebuttal. There is a tradeoff space here, which I think could be emphasized more - CCP requires datapath modifications and a new CC API, while ALCC requires no modifications to anything, and presumably your API is more similar to traditional per-packet implementations. From my reading, ALCC is basically saying, if I am in this specific cellular environment where the datapath is set in stone, you are offering an easier path to deployment than waiting for datapath features.”

In this document, we first highlight the main improvements we made in the submission to Eurosys 2021. Then, at the end of the document, we list all the reviewer comments we received earlier, along with our response to each of these comments.

1 Summary of improvements

In this section, we provide a summary of improvements we made to Eurosys 2021.

- Added a new section discussing how to integrate ALCC into Android phones for uplink. In addition to a preliminary ALCC Java library for Android.
- Enhanced the positioning of ALCC and extended the discussion on the comparison of ALCC and CCP.
- Reached out to CCP authors and got their opinion on how ALCC is different from CCP.
- Extended the description on the congestion control interactions between ALCC and the underlying TCP stack.
ALCC: MIGRATING CONGESTION CONTROL TO THE APPLICATION LAYER IN CELLULAR NETWORKS

Anonymous authors
Paper under double-blind review

Abstract
TCP is known to perform poorly in cellular network environments. Yet, most mobile applications are explicitly built on the conventional TCP stack or implicitly leverage TCP tunnels to various cellular middleboxes, including performance-enhancing proxies, application-specific edge proxies, VPN proxies and NAT boxes. Despite significant advances in the design of new congestion control (CC) protocols for cellular networks, deploying these protocols without bypassing the underlying TCP tunnels has remained a challenging proposition. This paper proposes the design of a new Application Layer Congestion Control (ALCC) framework that allows any new CC protocol to be implemented easily at the application layer, within or above an application-layer protocol that sits atop a legacy TCP stack. It drives it to deliver approximately the same as the native performance. The ALCC socket sits on top of a traditional TCP socket. Still, it can leverage the large congestion windows opened by TCP connections to carefully execute an application-level CC within the window bounds of the underlying TCP connection. This paper demonstrates how ALCC can be applied to three well-known cellular CC protocols: Verus, Copa, and Sprout. For these protocols, ALCC can achieve comparable throughput and delay characteristics (within 3-10%) as the native protocols at the application layer across different networks and traffic conditions. ALCC allows a server-side implementation of these protocols with no client modifications and with zero bytes overhead. The ALCC framework can be easily integrated with off-the-shelf applications such as file transfers and video streaming.

1 Introduction
According to Cisco's recent Global Mobile Data Forecast Update [14], approximately 50% of the global data traffic is generated by mobile devices. Mobile applications primarily rely on TCP as the basic transport protocol, and TCP is known to perform poorly in cellular networks [15, 46, 47]. Packet losses are frequent events over cellular networks due to varying link rates, fading, burst scheduling, and unpredictable user mobility. TCP congestion control responds poorly to such losses, unnecessarily reducing the sending rate by half, thus sacrificing valuable network bandwidth. To cope with the above issue, cellular network operators often significantly over-provision their network buffers (at least 10x the bandwidth-delay product) and rely on lower layer retransmissions to shield the end-to-end TCP connection from these losses. But this is known to result in bufferbloat [19], a phenomenon in which end-to-end packet delays are very high due to the large buffers being kept full most of the time. Over the past decade, several new congestion control protocols have been proposed to replace TCP [4, 15, 46, 47]. Despite their superior performance over TCP, none of these solutions have gained wide deployment over real network conditions. Some factors limit the adoption of these new solutions:

- Cellular networks adopt specific packet-filtering and packet-shaping middleboxes that often limit the type of protocol traffic allowed in the network (e.g. UDP packets may be blocked), thus forcing algorithms to use TCP.
- Most mobile applications are built using advanced API calls, which explicitly establish HTTPS tunnels to specific cloud services or secure Remote Procedure Call (RPC) to middleboxes. Both of these use the standardized TLS/SSL layer, tightly coupled with the underlying TCP layer [38], which again makes it difficult to circumvent TCP.
- New congestion control protocols need to be ideally enabled/installed during Operating System (OS) upgrades, which are infrequent due to slow-release cycles and mainly due to their potential interruption in existing settings.

Elaborating on the first two points, there are many different scenarios that illustrate the fact that an underlying TCP connection at the edge is unavoidable: i) server-client TCP connection that goes through a NAT proxy in the cellular network; ii) a secure server-client connection that traverses an active middlebox that performs header inspection and HTTPS-specific operations; iii) a server-client connection that traverses a middlebox that splits TCP connections [20, 37]; iv) a mobile transfer service using a cloud service API where the traffic between the client and server is explicitly routed via the cloud service; and, v) communication using non-standard socket interfaces to transmit/receive packets. In all these scenarios, there is an underlying TCP connection between the mobile client and a middlebox at the edge of the cellular network. These connections suffer from the bufferbloat problem due to the excessive buffer provisioning by the cellular
providers. TCP tunnels are very commonly used for: (i) mobile devices that are behind a NAT/Proxy [10]; (ii) Evolved Packet Core (3GPP) for supporting cellular networking functions; (iii) Third-party applications/services built for using secure tunnels to middleboxes. Mobile applications and services have no control and visibility over the underlying TCP tunnels and are exposed to the standard send/recv interface as in a TCP connection. This illustrates an ossification of TCP in the context of mobile communications.

Another problem that plagues existing solutions to congestion control is that of implementation. Most solutions require kernel modifications and re-implementations depending on the underlying datapath [31]. The existing brittle development ecosystem that is dependent on TCP has forced new congestion control protocols to be primarily built on top of UDP, such as QUIC [22]. However, any application requiring reliability or security support will need substantial code changes to integrate with these protocol stacks. Moreover, QUIC is not designed to address the bufferbloat problem in mobile environments and also suffers from packet reordering issues in cellular networks [36]. Congestion Control Plane (CCP) [31] is another solution to implement algorithms at the user level, but that design forces developers to rethink the congestion control implementation completely a different hurdle for developers. The design requires splitting up a CC algorithm into two segments, viz. a user-level agent provides a flexible execution environment in user-space for congestion control algorithms and a component that executes on the datapath itself. We compare our approach to CCP and discuss the pros and cons in greater detail in Section §2.1.1.

And finally, existing solutions do not provide options to implement CC algorithms quickly over application layer protocols such as HTTP and HTTPS. Again, due to the widespread deployment of such protocols over TCP, any framework that provides this capability would enhance the ease of implementation of new CC algorithms. Thus, this paper addresses the following question: Can we provide a framework for deploying new congestion control algorithms easily in mobile devices at the user-space that leverage the broad deployment of TCP and operate on top of the TCP and application-layer protocol stacks while offering similar performances as their native kernel implementations? Such a framework would be able to facilitate the rapid innovation, deployability and evolution of new protocols for mobile applications at the application layer and above, without modifying any aspect of the conventional cellular network architecture.

To address this question, we present Application-level Congestion Control (ALCC), a framework that executes CC protocols at the application-level to achieve similar performance as the native protocol while operating on top of an underlying TCP stack with multiple applications support. The ALCC framework enables careful packet pacing at the application layer regardless of the underlying TCP congestion window, thus limiting the traffic sent down to the TCP stack and thereby enforcing the sending rate of the application-layer congestion control protocol. Most TCP variants tend to ramp up to larger windows while significantly driving up the packet delays in cellular networks. ALCC constrains the application-level sending window to reduce bufferbloat in the system and maintains a low end-to-end packet delay. In the TCP tunneling context, ALCC specifically addresses the cross-congestion control interaction between the user-level CC and the underlying CC of the TCP tunnel in two ways: (a) When the underlying transmission layer socket sends a blocking signal, ALCC uses this signal to control the transmission rate; (b) The blocking signal also affects the behavior of the higher layer protocols to adapt to the varying signal.

ALCC solves many of the issues highlighted earlier. Since it makes use of existing TCP stacks, it is quickly deployable on a large scale. It provides a quick solution to implementing new or existing algorithms without significant effort. ALCC can also be implemented over application layer protocols on mobile devices, thus making deployment on mobile devices significantly easier than before. Another essential advantage of ALCC is that it can support specific protocols like Verus and Copa with server-side integration alone without any client-side modifications. For protocols like Sprout that require receiver feedback integration, ALCC needs to incorporate client-side changes. ALCC also operates with a zero-byte header. ALCC exposes the same TCP Berkeley socket APIs to the application layer, making it very easy to integrate into existing applications. For recently developed CC protocols [4, 46, 47], we demonstrate how easily these protocols can be blended into the ALCC framework and maintain the application layer congestion window in contrast to the underlying TCP congestion window. We show that these protocols within the ALCC framework imitate the original protocols’ performances and attributes while sustaining comparable throughput and packet delay characteristics (within 3 − 10%) irrespective of the underlying TCP flavors.

The paper makes the following key contributions:

- A framework to implement CC protocols within or above the application layer that sits on top of the legacy TCP stack. This facilitates rapid innovation, deployability and evolution of new protocols for mobile applications.
- Show how new CC protocols may be integrated easily into ALCC (with minimal code changes) and demonstrate through rigorous testing that these protocols achieve the same performance as their native implementations.
- Integrate the ALCC framework into existing off-the-shelf real-world applications such as the Bftpd FTP server.
- Show how for specific protocols, ALCC can support server-side integration without the need to modify the applications’ client implementation. It also does not add any additional overhead (zero-byte overhead).
2 Why ALCC?

We motivate the need to create a configurable framework for implementing new CC algorithms in the userspace on top of the conventional TCP substrate exposed by cellular networks.

Middleboxes, Tunneling, and API Gateways: Cellular network operators have deployed various types of middleboxes to make efficient use of their network resources and provide end-to-middle-to-end security for potential threats. This ubiquity has led to middle-box ossification, making them key control points in the cellular network architecture. Many of these middleboxes explicitly break the end-to-end connectivity model and support split TCP connections implicitly or explicitly [20]. As a result, many applications that run on cellular networks are implicitly tied to an underlying TCP connection, which runs on legacy TCP software, which is hard to change easily.

Recent years have seen a massive expansion in tunneling protocols, enabling the creation of Virtual Private Networks (VPNs), providing the illusion of a physical network to the user. Many users worldwide resort to mobile VPN clients to bypass censorship or access geo-blocked content, and more commonly, for privacy and security reasons. Even thoughVPNs encapsulate messages to traverse middleboxes, the encapsulation tunnel is terminated at the VPN server. The last hop from the VPN server to the desired remote host/server will continue to rely on the host’s TCP variant, thereby negating performance gains. Furthermore, the end-to-end CC is disrupted, and transport over encrypted tunnels may not allow other network entities to participate in CC. Finally, many mobile applications leverage API gateways that rely on HTTP variants and AMQP-like interfaces [2], essentially relying on an underlying TCP substrate to a gateway node. Netflix [29] and Amazon [1] are well-known public services that have adopted such API gateways.

Large delays in cellular networks: Mobile applications over TCP are known to experience considerable delays due to the complexity of the underlying cellular architecture. Recent cellular architectures [7] are known to employ large buffers to protect against packet losses.

This issue is well studied in prior work [11, 46, 47], which demonstrates that all known variants of TCP suffer from extensive delays in cellular networks since they aggressively set a large congestion window leading to excessive buffering at the base station and the gateway nodes. This is a crucial element of the motivation for designing a framework such as ALCC since ALCC is designed to throttle the sending rate in the underlying TCP connection thereby significantly reducing bufferbloat and packet delays and improving overall performance.

2.1 ALCC vs. popular related frameworks

Since the in-kernel implementations of congestion control protocols is a challenging task, especially at scale, many new congestion control protocols require useful libraries not supported by the kernel (i.e., libboost and alglib in Verus [47], or Bayesian forecasts in Sprout [46]). These rely on implementing these protocols within the userspace over UDP instead. UDP, however, lacks the required security support that is needed by many applications. Besides, many firewalls and middleboxes are configured to drop UDP traffic due to the lack of congestion control or explicit connection setup/tear-down. This hinders the deployment aspects of the implemented UDP-based congestion control protocols. In contrast, TCP does not suffer from this problem, making it the perfect candidate for deployment. Hence, a possible solution that could combine the deployment benefits of TCP while allowing better congestion control logic is highly desirable.

ALCC leverages the kernel’s TCP implementation while allowing developers to implement their congestion control logic within the userspace, effectively controlling the data flow down to the TCP stack and enforcing the application-level congestion logic to dominate TCP’s default congestion control. ALCC shares similar goals as other popular frameworks in literature today, such as Congestion Control Plane (CCP) and Google’s QUIC. Next, we will discuss the main differences between ALCC and these frameworks. Table 1 shows high-level comparisons to QUIC and CCP.

2.1.1 ALCC vs. CCP

ALCC, in spirit, shares some of the design goals of CCP [31], in the sense of providing developers with a way to easily implement their congestion control protocol within the userspace, thus enhancing the pace of development and ease of maintenance of congestion control algorithms. However, ALCC addresses a fundamentally different problem relevant in cellular networks, namely that the middleboxes in cellular networks may use TCP tunnels and split TCP connections for performance reasons. In these scenarios, by throttling the traffic through the TCP connections, ALCC can reap significant benefits in performance. CCP, on the other hand, is a generic framework for implementing congestion control algorithms in the userspace. ALCC leverages the kernel TCP implementation without the additional complexity of directly modifying the data. This allows ALCC to benefit from the wide deployment popularity of TCP.

Like CCP, ALCC also has a kernel component that shares flow-level information (such as end-to-end delay, bytes in flight, etc.) as well as information from each TCP ACK received (such as sequence number, bytes acked, etc.) with the userspace program so that the CC algorithm can make use of them. However, when using CCP, every CC algorithm needs to be rewritten entirely. Algorithms must be implemented over two pieces: i) datapath logic, and ii) the actual CC logic.
The datapath logic is a small piece of code in LISP-like syntax that specifies which variables from the kernel datapath need to be reported to the CC algorithm at what temporal granularity, while the actual CC logic to control the sending window is implemented in Rust or Python. This type of implementation requires a rethinking of the CC algorithm, which adds an additional burden on the developers. In contrast, ALCC maintains a straightforward framework that does not require CC algorithms to be re-implemented but rather easily ported by replacing native socket function calls (such as send() and recv()) with calls to the corresponding functions exposed by the ALCC library implemented in C++.

And lastly, ALCC is much more amenable to a mobile implementation than CCP. This is due to several reasons: i) CCP requires a specific kernel patch to the Linux kernel that enable reporting of variables from the network datapath to the application layer; ii) the overhead of using CCP has not been studied in mobile devices, and the CC algorithms implemented in CCP need to be running in the background as a user process; iii) CCP currently only has libraries for implementing congestion control algorithms in Rust and Python, and porting to Java (Android) or Objective C (iOS) or others is a significant implementation hurdle. In contrast, ALCC requires far fewer modifications and less setup time. Even if an application’s source cannot be modified, calls to native socket send/recv functions can be redirected to the ALCC interface without significant effort.

<table>
<thead>
<tr>
<th>Features</th>
<th>QUIC</th>
<th>CCP</th>
<th>ALCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Congestion Control</td>
<td>Userspace implementation</td>
<td>Userspace implementation</td>
<td>Userspace implementation</td>
</tr>
<tr>
<td>Supported Transport</td>
<td>UDP</td>
<td>Any data path including UDP &amp; TCP</td>
<td>TCP only</td>
</tr>
<tr>
<td>Reliability</td>
<td>Externally configurable</td>
<td>Externally configurable</td>
<td>TLS only; M-TCP dependant (not configurable)</td>
</tr>
<tr>
<td>End Hosts Support</td>
<td>Mandatory Client &amp; Server implementation</td>
<td>Supports Client only, Server only, or both implementations</td>
<td>Supports Server only or both Client and Server implementations</td>
</tr>
<tr>
<td>Re-transmission Mechanism</td>
<td>QUIC’s packet number with directly encoded transmission order</td>
<td>- Based on TCP sequence numbers (ACKs)</td>
<td>Based on TCP sequence numbers (ACKs)</td>
</tr>
<tr>
<td>Design</td>
<td>- Crypto handshake to minimize setup RTT</td>
<td>Control Plane agent in the userspace</td>
<td>Dominant userspace congestion control loop running atop TCP</td>
</tr>
</tbody>
</table>

The aim is to find a way for a new CC algorithm to interact with existing legacy TCP stacks in a controlled manner. The goal is to get the advantages of both the new CC algorithms and the widespread deployment of the legacy TCP-based underlying architecture. We achieve this by leveraging TCP’s buffers and using them to mimic the CC algorithm’s sending behavior in the application layer.

3 Application-Level Congestion Control

To address the deployment challenges of recently proposed cellular CC protocols, we offer Application-level Congestion Control (ALCC) as a framework to execute these protocols at the application layer without modifying the underlying transport layer. ALCC explores the question: Can we derive the performance benefits of new CC algorithms by deploying them at the application layer on top of traditional TCP stack?

Table 1: High-level comparison of QUIC, CCP and ALCC frameworks
3.1 Flying under the TCP radar

Each of many recently proposed protocols such as Sprout [46], Copa [4], and Verus [47] claim that their CC responds more efficiently in cellular network environments than existing protocols. Yet, despite their superior performance in a cellular context, actually deploying these protocols at scale remains a challenging task.

We suggest migrating these new CC protocols to the application layer to operate over the widely used TCP stack (without any modifications to TCP). To better understand how this is achieved, let us first explore the following hypothesis: the effective congestion window can be actively controlled from the application layer (i.e., pacing application data lower than TCP’s CWND). It is well known that the widely used TCP flavors, such as Cubic and Reno, maintain unnecessary large CWNDs and cause high packet delays. We evaluate two experiments that aim to test the following hypothesis: is the TCP CWND affected by the amount of data sent from the application layer? Does it still maintain a large CWND even if there is not enough data to exploit this large CWND fully? If this hypothesis holds, then the application layer can efficiently send data without incurring high packet delays or causing congestion/packet loss. This is achieved by throttling the application data flow to keep it below TCP’s CWND. The control done in the experiments is achieved by limiting the application data flow sent down to the transport layer (we have chosen the standard TCP Cubic [24]). Unless otherwise stated, the tests were conducted on an experimental Mininet testbed consisting of a fixed bandwidth link of 12 Mbit/s, a client, a server and a router. In the first experiment, we implemented a shim layer within the application layer that maintains a static congestion window set to the theoretically required window to saturate the network link. In the second experiment, we implemented the shim layer to randomly choose the congestion window every second. Both experiments were tested on Ubuntu 18.04.1 with kernel version 4.15.0. Figure 1 compares the congestion windows of these two simple application layer protocols with the underlying TCP Cubic window. The solid lines represent the application layer window, and the dashed lines are the TCP Cubic congestion window. The figure confirms our earlier hypothesis that the TCP Cubic window is unnecessarily high, even higher than the application layer window. We made similar observations for TCP Reno and Bic. The bottom figure highlights the fact that we can arbitrarily control the sending window within TCP Cubic’s envelope.

Figure 2 shows how the application layer can control the TCP Cubic congestion window to achieve better performance. Here, the application layer maintains a static congestion window and is compared against a legacy TCP Cubic network stack. We observe that the legacy TCP Cubic congestion window increases significantly over time, causing high packet delays. In contrast, controlling the window at the application layer (while still running on top of the TCP stack) achieves similar throughput without causing high packet delays. These results tentatively prove the hypothesis we discussed earlier by highlighting that one can perform a second CC loop within the application layer while leveraging TCP’s limitation of maintaining a large congestion window. This motivates the idea of running CC protocols within the application layer without having to replace the TCP stack.

3.2 Overview of ALCC

The core idea of ALCC is to perform CC at the application layer by staying under the TCP radar. ALCC’s primary function is to replace the congestion window of TCP with the congestion window computed at the application layer by a new CC protocol. Based on this calculated window, ALCC tightly constrains the application data flow to the transport layer; the congestion window of ALCC is the minimum of the window size computed by the new CC protocol and the one of the underlying TCP protocol.

In essence, the ALCC framework emulates the cellular CC
protocol at the application layer and computes the transmit window of the protocol. ALCC relies on the TCP socket interface feedback to implicitly learn the transmit window of the underlying TCP variants’ protocol. For every packet transmission, if the TCP socket reports a full buffer and blocks on a potential transfer (or indicates full buffer in a non-blocking socket), ALCC delays the next packet transmission. This blocking signal serves as an indication from the underlying TCP stack that the network might be congested and as a result the ALCC protocol needs to slow down.

The basic version of ALCC maintains a separate layer of acknowledgments at the application layer to keep track of the number of outstanding packets at the application level. This enables the ALCC layer to determine the packet transmission rate at the application level. Similar to Verus and Copa, ALCC can also support a server-side implementation where only the server can execute the CC protocol, and the client runs a native TCP connection with no application layer CC. In this case, the server needs to rely on TCP layer acknowledgments to track the outstanding window of packets.

The simple design of ALCC allows it to transmit at a lower rate than the TCP window and correspondingly achieve better delay characteristics. Surprisingly this strategy enables ALCC to achieve similar delay-throughput trade-offs as the native CC protocols while maintaining the fairness properties of TCP. By controlling the sending window, ALCC can reduce network buffer sizes and hence end-to-end delays without sacrificing throughput in comparison to standard TCP.

4 Realizing ALCC

The ALCC framework is implemented as a C++ library that acts as a shim layer connecting the application layer and the transport layer\(^1\). In other words, the ALCC C++ library is implemented in the userspace as a wrapper around the default Linux Berkeley sockets TCP implementation. It is designed to expose the same socket API to the application layer. This is performed to facilitate a smooth and easy integration of the ALCC library into existing applications. We describe three different implementations:

- **Server-side ALCC library**: This is the default ALCC implementation, which relies on the native TCP acknowledgments instead of the implemented CC protocol’s acknowledgments. Here, the client is kept unmodified. This is realized by an ALCC kernel component that is implemented as a Linux kernel module. The ALCC kernel module is implemented with Netfilter and NetLink, and it acts as a cross-layer module to filter TCP acknowledgments and send them to the userspace program.

- **Client/Server ALCC library**: This is a special implementation extended from the above, without the use of the kernel module. It relies on the implemented CC protocol’s acknowledgments, and the client code is also required to use the library to send back acknowledgments. This implementation is meant for protocols that rely on external signals apart from acknowledgments, where some additional data needs to be shared by the client to the sending process.

- **Mobile Java ALCC library**: This implementation is meant for Android mobile phones that allows them to use ALCC to send data efficiently in the uplink direction. This implementation is similar to the Client/Server ALCC library, where it does not rely on a kernel module but rather implements its own acknowledgment mechanism.

A significant benefit of the default Server-side ALCC library is its single-side (server only) modification, which does not require any client changes. This is an exceptional advantage because it simplifies the deployment significantly, where ALCC relies on the underlying TCP stack for packet sequence numbers and acknowledgments. This is perfect for supporting the implementation of CC protocols such as Copa and Verus in the downlink direction, where TCP’s acknowledgments would suffice. On the other hand, CC protocols such as Sprout do require additional information to be sent back to the sender from the receiver in addition to the acknowledgments. This is why we implemented the second library to deal with integrating these protocols. For example, Sprout’s receiver sends back the observed packet arrival times as the primary signal to determine the network condition.

4.1 Server-side ALCC library

4.1.1 ALCC Userspace Module

The userspace module is where the core part of ALCC is implemented; it is responsible for executing the application-layer CC protocols. We will call the ALCC userspace module as the “ALCC library”. The application uses the ALCC library to open an ALCC socket instead of a TCP socket. The ALCC framework of the library is shown in Figure 3a. The main philosophy of the ALCC library is to provide placeholder functions to integrate any CC implementation easily. The idea is to split the CC implementation into three processes: i) basic CC logic, ii) sending-related functionality, and iii) receiving-related functionality. The sending mechanism of ALCC also opens a standard TCP socket to send data down to the transport layer. The ALCC library is implemented as a C++ class, where the core part of the implementation consists of a:

1. Circular queue implementation
2. Sending thread (ALCCSocket::pkt_sender)
3. Receiver thread (ALCCSocket::ack_receiver)
4. CC logic thread (ALCCSocket::CC_logic())

The circular queue is used to store the data sent down from the application layer so that ALCC can pace the MTU-sized packet sending based on the CC sending mechanisms. Introducing an intermediate queue allows ALCC to leave the sending mechanism of the application unchanged. Like the standard TCP send buffer, the ALCC framework blocks the application once the intermediate queue is full, which may occur if the underlying TCP sending kernel buffer is full.

![Diagram of ALCC infrastructure](image)

**Figure 3: ALCC library framework architecture**

The sending thread is responsible for sending data packets from the ALCC queue down to the TCP stack using a regular TCP socket. It paces the data sent down by relying on the CC logic thread’s decisions to regularly determine the so-called “application layer congestion window” (analogous to the transport layer congestion window). The application layer congestion window is computed by the CC protocol running within the ALCC library. The receiver thread implements the CC protocol logic when receiving an acknowledgment. As stated earlier, ALCC does not implement its own acknowledgment but rather relies on the underlying TCP ACKs. These ACKs are sent by the ALCC kernel module running within the Linux kernel. For the application layer, ALCC exposes two function calls: alcc_accept() and alcc_send() that are meant to replace TCP’s accept() and send() functions. These are the only modifications required at the application to use ALCC. The new function calls are intentionally kept similar to the default TCP ones—in terms of the arguments they use—to ease the integration as:

```c
int alcc_accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
int alcc_send(int sockfd, const char *buffer, int buflen, int flags)
```

Alternatively, we can also simply use the `LD_PRELOAD` [21] trick to change the TCP socket system calls accept() and send() to the ALCC calls, which makes ALCC even more usable since there won’t be a need to replace the system calls within the applications.

Within the alcc_accept() function, the basic TCP accept function call is performed, and the corresponding TCP socket is passed to the ALCC framework object instantiation. This socket is then used by the ALCC object to send and receive data. As for the alcc_send() function call, it mainly accepts the data from the application and then inserts it to the ALCC internal buffer to be sent later by the ALCC sender thread.

### 4.1.2 ALCC Kernel Module

A couple of recent delay-based CC protocols, such as Verus and Copa, rely on their own acknowledgments to infer what is happening in the network. Their native implementations rely on the UDP protocol as the underlying transport protocol. With the introduction of the ALCC library, and because ALCC runs on top of TCP sockets, an opportunity arises to simplify the integration efforts in real-world applications. Some of the ultimate design goals behind any successful protocol are simplicity and efficiency. To achieve a zero-byte overhead and server-side only modifications, we modified the Verus and Copa implementation within ALCC to rely on TCP ACKs rather than on their own. In our implementation, the TCP ACKs are retrieved through an ALCC kernel module that we built using the Linux kernel framework known as Netfilter [5]. Netfilter offers packet manipulation via various hooks into the network layer. We have used the NF_IP_PRE_ROUTING hook, triggered by any incoming traffic soon after entering the network stack, and before the kernel performs any routing choices for packet sending. Figure 3b shows an example of multiple components fitting together to provide an insight into how filtering and communication between the kernel module and the userspace module are achieved. When an IP packet arrives at the network layer, the kernel sends the packet to the Netfilter module, which then transfers it to the iptables module. The latter holds a set of rules defined by the ALCC kernel module to specify the actions to be taken when the desired packet is detected. It first inspects the transport layer type within the IP packet. In the case of TCP, it extracts the TCP header and checks the destination port number. If an ALCC socket has already registered that destination port, the hook function extracts the ACK details and sends it to the respective ALCC userspace process. For the kernel-to-userspace delivery, the ip_queue module uses Netlink sockets. We have implemented a signaling protocol between the ALCC userspace module and the kernel module. When a new application opens an ALCC socket, the ALCC socket first opens a legacy TCP socket, which gives back to the ALCC framework the actual port number used with this socket. The ALCC userspace module would then send a `port_registration` message to the kernel module to register its process id with the associated port number. The
ALCC kernel-module maintains a mapping table between the different port numbers and their corresponding ALCC userspace process IDs. When the ALCC flow is finished, it sends a port_release message to the kernel module to remove the registered port number. When CC protocol designers use the ALCC library to integrate/implement their protocol, there will be no requirement to implement any functionality within the ALCC kernel module.

4.1.3 CC protocols integration into ALCC

To evaluate the ALCC framework’s performance, we ported two protocols into the ALCC library – Verus and Copa. In addition to the significant benefits of running a CC protocol within the application layer, the time and effort taken to integrate such a protocol are equally crucial.

**Verus integration:** The Verus integration was straightforward since its threads fit well in the structure of the ALCC library class. Verus has four main threads: a sending thread, a receiver thread, a logic thread, and a delay profile thread. When it came to porting the Verus code base into ALCC, one of the main changes was integrating the receiver thread and the Verus acknowledgments. Verus uses a header that consists of a sequence number, CWND when the packet was sent and the sent time. Since the main target of ALCC was to simplify the integration effort and keep it bound only to the server-side, we relied on the underlying TCP’s sequence numbers. We also created two different mappings within the ALCC library to store both the congestion window at the time the packet was sent and the sent time (so as not to carry it with the header). Two main challenges in relying on TCP’s ACKs were obtaining the sequence numbers at the application layer (since TCP runs at the kernel), and TCP’s sequence numbers are bytes-offset. In contrast, Verus’s sequence numbers are simple integers. The first challenge was solved using the ALCC kernel module. The second challenge was solved by maintaining a mapping between TCP sequence numbers and the sequence numbers of Verus. During the initial phase, ALCC must listen for the first sequence number exchanged between the client and the server during the TCP handshake, since TCP chooses the starting sequence numbers at random. The Verus code base took about a day to port into ALCC.

**Copa integration:** We leveraged the generic CC implementation of Copa [3]. The main challenge was that Copa’s code implements four different CC protocols: Copa, Remy, kernel CC (Cubic on Linux), UDT’s [23] TCP AIMD implementation and PCC (deprecated). All of these protocols, including Copa, were implemented using UDT’s class. The challenge was to extract Copa’s codebase and any additional required code from the UDT class. Luckily, the main Copa’s implementation was bounded to the markoviancc.cc and markoviancc.hh files. As for the main logic of the sending/receiving packets we extracted the code from the ctcp.hh, specifically CTCP<T>::send_data function. This function handles the sending of packets and then checks if any data is pending to be received. We had to split the function into two halves, where the sending code was moved into ALCC’s sending thread, and the packets’ acknowledgments handling logic was moved into the receiver thread. We had to duplicate some of the variables that were used by both parts. We faced some challenges within the receiver logic because TCP is a byte stream protocol where TCP sometimes acknowledges multiple packets in a cumulative acknowledgment. Since Copa’s receiver logic was built to handle a single packet acknowledgment, we first figured out how many packets are being ACKed and then wrap Copa’s receiver logic with a loop that can handle the multiple acknowledgments. Additionally, Copa uses a map for unacknowledged_packets, which before was protected since the sender and receiver logic were part of the same function and were executed one after the other. Due to the split in the ALCC framework, we had to protect this map from corruption by simultaneous access by both threads, which was achieved using mutex locks. Like Verus, we had to add mapping to store Copa’s packet sent times to compute the packet delays. We also had to handle TCP’s byte-offset sequence numbers and map them to Copa’s integer-based sequence numbers. The porting of Copa’s code-base took about two working days.

4.2 Client/Server ALCC library

This library implementation is very similar to the above implementation, except for the following key aspects. First, this library does not require the ALCC kernel module since it relies on the acknowledgments sent by client-side CC protocol implementation rather than the native TCP ACKs. Second, the ALCC userspace module is almost identical to the server-side library implementation, except for how the acknowledgments are handled. The ALCC userspace sender module relies on the acknowledgments that the client will be sending back to the application. In the case of the client-side modifications, the ALCC library would modify the TCP receive function call to first read the CC protocol header from within the TCP socket and then send back an ACK as a separate packet to the server. Then the CC protocol packet payload is read and returned to the client app to be used as useful application data. Of course, here, due to the byte-stream nature of TCP, the CC protocol payload length is determined from the packet header. We have modified the client side to send the packet delivery forecast back to the sender in the Sprout integration.

4.3 Implementing ALCC on a mobile device

Mobile OSs such as Android impose restrictions on modifying the underlying TCP stack, such as not providing access to kernel modules. To circumvent these, we have developed a preliminary Java library that supports running ALCC atop application layer protocols such as HTTP. This approach has
multiple advantages, such as maintaining the TLS/SSL connections, as the underlying TCP implementation is untouched or transparent to (reverse) proxies that might intercept the connection along the way. The library uses the chunked transfer encoding in the case of HTTP/1.1 [17], and streams in the case of HTTP/2 [6] to chunk the data while maintaining a persistent TCP connection between the client and the server. The main task of the library is to send the chunked data packets as separate HTTP packets/frames and receive acknowledgments from the server in the form of an HTTP responses to obtain round-trip time estimates. To implement the library, we used the open-source project OkHttp [39] which is part of the Android suite since Android 5.0. Unlike the desktop Linux implementation which uses a kernel module for cross-layer communication, in this Android library, we implemented our own acknowledgement mechanism by adding a custom header with sequence numbers, thus obviating need for a kernel module. We also extended the ALCC server library to send back ALCC acknowledgments with the sequence numbers. This library would be handy for video conferencing or live streaming social media applications on a mobile device.

To ease the integration of CC algorithms within the Java Android ALCC library, one possible approach is to utilize Model-Driven Interpretable (MDI) congestion control [27]. MDI allows approximating any congestion control algorithm as a general discrete-time Markov model over a 2-dimensional state space, represented in the form of a state-transition probability matrix for that algorithm. Each state is a tuple of the relative change in the network delay and the sending window size. The matrix describes transition probabilities between every pair of states, and is obtained by training the algorithm on a large set of network configurations. MDI versions of popular algorithms have been shown to be mimicking the actual algorithms in terms of throughput and delay characteristics on real traces. Thus, using the publicly available Markov models trained independently over diverse network conditions, we can easily and effectively replicate the behavior of many algorithms atop ALCC on Android.

### 4.4 Integration into off-the-shelf Applications

In this section, we discuss and highlight the integration efforts of the ALCC library into existing off-the-shelf applications. To demonstrate that the ALCC library can easily be integrated into real applications, we chose a number of off-the-shelf applications as a proof-of-concept examples for the integration: FTP using the Bftpd server [41], web using the Ryuuk concurrent web server [30], and video streaming using the RTMP Server for Adobe Flash player [28].

First, we started integrating ALCC into the bftpd server application. Since easy integration was one of the main motivations behind designing the ALCC framework, we exposed three different ALCC functions to act as a replacement for their TCP counterpart within the application implementation, these are: alcc_accept(), alcc_send(), and alcc_close(). We searched within the bftpd application for the location where they instantiate the TCP socket, mainly looking for the TCP accept() call, which we found inside the command.c file. We then simply replaced the accept() call with alcc_accept(). The latter is implemented to instantiate the ALCC object, which would create the ALCC queue and the multiple ALCC threads. We then replaced the send() function call with our own alcc_send(). However, due to the byte-stream nature of TCP, we had to enclose the sending function in a while loop that can guarantee the complete sending of all the required bytes. Finally, we had to replace the TCP close() call with the alcc_close() call so that we can make sure that all data stored within the ALCC queue are sent first before terminating the connection. For the RTMP Server and the Ryuuk web-server integration, we had to do the same as above. We replaced the above three TCP function calls with their ALCC counterparts. That is mainly found inside the main.cc file for the RTMP Server, and the SocketListener.cpp and SocketStream.cpp for the web-server.

### 5 Evaluation

Our evaluation demonstrates that the ALCC framework, with three different integrated CC protocols (Verus, Sprout, and Copa), can achieve the same throughput and delay distribution characteristics as their native protocols regardless of the underlying TCP transport protocol. The evaluation were conducted on an Ubuntu 18.04.1 machine with kernel version 4.15.0, with Intel Xeon(R) CPU E3-1246 v3 @ 3.50GHz x 8 with an 8GB RAM.

Applications that run on top of the ALCC stacks would naturally observe similar behavior as the native protocol under different network conditions. The main goal of our evaluation is to demonstrate how the integrated CC protocols within the ALCC framework closely match their native protocol performance including their temporal characteristics. To demonstrate reproducible results and control for different aspects of the evaluation under the same network conditions, we collected a diverse set of cellular network traces and used the network emulation environment Mahimahi [32] that enabled us to test different protocol implementations across different network conditions in a controlled manner. Our ALCC implementation runs over real mobile networks, and we have conducted several tests running various off-the-shelf applications with ALCC over 4G networks.

---

5.1 Channel Traces

We compare the performances of Verus, Sprout and Copa over different TCP variants (Cubic, Bic, and Reno) within the ALCC framework. The experiments are conducted using the Mahimahi network emulator with various channel traces, some taken from published papers and others recorded from real cellular networks. 4G Verizon: taken from [46] and represents a recorded channel over Verizon’s 4G network in the US. Rapidly changing network: inspired by [15], this trace represents a highly fluctuating channel, where the magnitude is varied randomly every 5 seconds. 3G Etisalat: taken from [47] and represents a recorded channel over the Etisalat 3G network while driving on a highway at 120 km/h.

Other channel traces are collected by setting up a server located at a University campus and four Android smartphones. A bi-directional setup was used to monitor the downlink and uplink channels using a 3G HSPA+ cellular network. Both the server and client concurrently send UDP packets of 1400 bytes. Data rates of 2.5 Mbps and 5 Mbps were set for uplink and downlink, respectively. However, these data rates do not necessarily indicate the maximum capacity of the cellular network. We assume that the channel is not over-saturated by using these data rates, and packet-buffering is minimized under perfect channel conditions. Measurements of three different scenarios are captured with varying properties of mobility. The scenarios are City Drive, Campus Walk, Highway Drive, and Beachfront Walk. The channel traces are generated from the packet arrival timestamps at the receiver and the inter-arrival times between consecutive packet arrivals. Additionally, the channel traces from all our four phones were combined into one large trace to emulate significant user contention.

5.2 ALCC-Verus vs. Verus

This section highlights the performance comparison of the ALCC-Verus implementation on top of legacy TCP Cubic stack and the native Verus. We examine the throughput and packet delay for both cases. The experiment is conducted using the Mahimahi emulator using multiple channel traces, with a bottleneck buffer size of 2 MB. We only show the results for four of the traces due to space limitation (City drive, 3G highway drive, Beachfront walk, and rapidly changing channel). Figure 4 shows the comparison of the achieved instantaneous throughput and delay over time for the selected four channel traces, where the upper part of each sub-figure depicts the throughput, and the lower part of the figure highlights the delay performance in logarithmic scale. The results show that both protocols deliver almost identical throughput and packet delays on all channel traces. It can be observed that ALCC-Verus inherits Verus properties, such as avoiding network buffer overfilling, while fully saturating the link capacity. Worth mentioning here that the ALCC-Verus used in these experiments ran on top of TCP Cubic.

![Figure 4: Instantaneous throughput and delay over time](image)

To accurately measure the performance similarity between the ALCC-Verus and Verus, we computed the Probability Density Function (PDF) for both the throughput and the delay using the Seaborn kernel density estimate [43]. These PDFs were calculated over 20 independent runs for each channel trace to obtain statistical significance. Figure 5 shows the PDFs for the selected channel traces, where the above part of each sub-figure shows the throughput PDF, whereas the lower part shows the delay PDF. From the comparison, it can be noticed that the ALCC-Verus PDF, depicted in red, does match the shape of the original Verus PDF shown in blue for all channel traces. Apart from some negligible marginal delays, variations are seen in the distribution of the rapidly changing network delay. Figure 6 shows an overall summary of the results comparing the different values of the results population. Each protocol is depicted by a circular shape representing the operational region of the protocol circumscribed by the 25% and 75% percentile of the obtained throughput and delay, where the crosses (x) indicate the median values. The lower and upper part of the shape represents the 25% and 75% of the throughput, respectively (y-axis) whereas the left and right part of the shape represents the 25% and 75% of the delay, respectively (x-axis). The results show that even though ALCC-Verus runs on top of TCP, it is still capable of achieving similar statistical performance in terms of delay and throughput with a minor delay penalty not exceeding 15% in the worst case scenario (i.e., rapidly changing channel).

5.3 ALCC-Sprout vs. Sprout

This section highlights the performance of the Client/Server ALCC library implementation using Sprout as the use-case scenario. We evaluated ALCC-Sprout over multiple other channel traces, and it has shown similar results to the one
discussed in this section. However, we have not presented them in the paper due to the page limitation.

Figure 7 shows the performance comparison of ALCC-Sprout vs. Sprout: this was performed using the rapidly changing network trace. The instantaneous throughput and delay results, depicted by Figure 7a, demonstrate that ALCC-Sprout matches the original Sprout performance, thoroughly saturating the channel capacity and having similar delay characteristics. This can be further proven by the PDF results shown in Figure 7c, where the ALCC-Sprout PDF matches the distribution of the native Sprout protocol. Finally, the population summary results show that the ALCC-Sprout version incurs a slightly higher end-to-end delay than the original version for both the 25% and 75% of the population. However, the difference over the median is marginal.

5.4 ALCC-Copa vs. Copa

Figure 8 shows the throughput and delay comparison of native Copa versus ALCC-Copa (running on top of TCP Cubic). We chose to show the results achieved over the following channel traces: 4G Verizon, 3G highway drive, Campus walk, and beachfront walk, performing 20 independent runs per trace with the same characteristics defined in the previous subsection. Although we show the results for these four channel traces, the other traces show similar performance and are omitted due to space restrictions. Similar to the ALCC-Verus results, it can be observed in Figure 8 that ALCC-Copa does achieve nearly equivalent instantaneous throughput and delay to Copa. The PDFs of the throughputs and delays of the two protocols are shown in Figure 9. It can be seen that ALCC-Copa achieves near-identical distributions to Copa. Figure 10 shows that the operational region of the protocols does match in all traces, despite some minimal difference in the highway traces. Moreover, from the median values, it is evident that ALCC-Copa achieves the same throughput, deriving all properties of the original Copa.

5.5 ALCC with Multiple TCP Flavors

In the previous sections, we have demonstrated that three different ALCC protocols achieve similar throughput and delay characteristics to their native protocols while operating on top of TCP Cubic. In this section, we investigate if the same holds if the ALCC protocols operate on top of other popular TCP variants such as TCP Bic, and TCP Reno. Figure 11 and 12 show the overall operating region results and the throughput and delay PDFs for both ALCC-Verus and ALCC-Copa, respectively. The results show a comparison to different underlying TCP flavors. These experiments were
conducted over all the channel traces; however, we show the results of two traces per protocol. The results confirm that ALCC does exhibit similar delay and throughput characteristics irrespective of the underlying TCP flavors.

**5.6 CPU utilization**

Benchmarking the CPU utilization is an essential aspect of evaluating the ALCC framework compared to the native CC protocols. We demonstrate the userspace CPU overhead—caused by ALCC—by running both Copa and Verus with and without the ALCC framework. This evaluation is done over a cellular network environment, utilizing the 4G Verizon channel trace in the Mahimahi emulator. TCP Cubic is used as the underlying transport layer protocol. We measured the average userspace CPU utilization for Verus and Copa (with/without ALCC) running over a server with Intel Xeon E3-1246 v3 Octa-core (8 Core) 3.50 GHz Processor. The results have shown an increase in the CPU utilization for both Verus and Copa, ranging from 1.5x – 2x compared to the native versions without ALCC. Similar observations were also reported by the authors of CCP [31], which is mainly caused by the Inter-Process Communication (IPC) overhead, which in ALCC is due to the cross-layer TCP ACKs being sent from the kernel ALCC module to the userspace ALCC module. Because ALCC runs only on the server-side, the increase in CPU can be considered a tolerable side-effect, given the overall benefits of ALCC. Additionally, the current ALCC implementation is not optimized to reduce the IPC and the cross-layer TCP ACKs sent from the kernel, further optimizations can reduce the CPU overhead significantly.

For the Java Android ALCC library, the CPU utilization should be significantly lower than the server-side implementation. In this library, we do not rely on a kernel module to get the acknowledgment. Given that native Copa or Verus can not operate on the native Android mobile in the uplink direction, phone measuring the CPU overhead was not feasible.

**5.7 ALCC under Packet Losses**

We further investigate how ALCC responds to random packet losses in a cellular network. We evaluate Verus as the primary congestion control in ALCC with Cubic as the underlying default TCP transport layer protocol. Moreover, the 4G Verizon channel trace is utilized in Mahimahi with an enforced 1% stochastic loss, a relatively high error rate in cellular contexts especially after lower-layer recovery within the cellular network [13]. Figure 13 shows the performance of ALCC-verus (running atop TCP Cubic) compared to the native Verus (implemented atop UDP) under the same channel settings. In the time window between 100-150 seconds, the ALCC-verus throughput is significantly higher compared to Verus. Surprisingly, we observe that the ALCC-Verus performance (with 1% loss) is comparable to the ALCC-Verus performance without the packet losses as shown in the Figure 13. An interesting observation in this scenario is that even though TCP Cubic and Verus would multiplicatively decrease the CWND window upon a packet loss event, the TCP CWND in Cubic is still large enough for ALCC to maintain its rough operating state and the status of its in-flight packets (i.e., its control over the sending rate) while Verus significantly reduces its rate due to the losses. This highlights a setting where ALCC-Verus can reach a higher throughput during packet loss events compared to the delay-based CC logic of Verus implemented on top of UDP. Given that ALCC does not aggressively ramp up the network buffer of the underlying TCP connection, ALCC indirectly reduces the chances of the underlying TCP connection experiencing a timeout with a large packet recovery period due to a buffer overflow event. Even under such set-
tions, Cubic is known to aggressively ramp up the congestion window after a timeout within a few RTTs, enabling ALCC to recover quickly.

6 Related Work

Over the years, numerous TCP versions have been proposed [8, 9, 12, 18, 24, 26, 40, 42, 44] to optimize transport protocol performance. The default TCP Cubic revises the additive increase, multiplicative decrease (AIMD) practice of TCP to promptly saturate the link in high bandwidth-delay product (BDP) networks. But Cubic is inefficient to trace time-varying wireless link capacities properly. However, delay-based congestion control protocols, such as TCP Vegas [8], have earned increased attention in the context of cellular networks due to its performance concerning non-congestion induced losses. Several other protocol designs focus on learning optimal performance, despite variations in the network environment. The essential approach is to search for actions directly to maximize throughputs and reducing delays. Remy [45] employs off-line training to achieve the optimal mapping connecting network conditions and the CWND adjusting function. In contrast, the performance-oriented congestion control (PCC) protocol [15] utilizes online learning to determine the sending rate for maximizing the value of a utility function based on feedback from the receiver in real-time. PCC Vivace [16], which followed after PCC [15], leverages ideas from online (convex) optimization in machine learning to do rate control while alleviating the bufferbloat problem. However, Vivace’s performance in LTE networks suffer due to noisy environments. 3G and LTE network measurements [25, 34] have demonstrated that variations in the physical properties of the radio channel can cause significant performance differences.

Due to highly variable channel fluctuations, cellular networks often use deep buffers, which leads to significant self-inflicted packet delays due to bufferbloat. Bufferbloat can be avoided by employing Active Queue Management (AQM) schemes like CoDel [33] and PIE [35], however, despite achieving low packet delays, these schemes suffer from under-utilization of link capacity. Without AQM, Cubic and NewReno rely only on packet drops as a sign of congestion. With deep packet buffers, this signal is too infrequent for active adaptation to varying link conditions. Protocols like Sprout [46] and Verus [47] overcome the sparceness of packet drops by utilizing RTT and send/receive rate information with prediction strategies to conclude accessible link capacity. Sprout [46] is designed for real-time streaming applications that demand high throughput and consistently low packet delays over cellular networks. Verus [47] is a delay-based congestion control protocol designed for highly fluctuating mobile networks. Verus makes decisions on changing delay profile curve over time and adapts to the instantaneous properties of the channel conditions. BBR [11], also proposed by Google has shown good results over cellular networks. BBR uses the round trip propagation time and bandwidth of the bottleneck link to find the optimum operating point for CC. UDP based data Transfer Protocol (UDT) [23] is a user-space framework that carries the protocol design, used for configuring and evaluating new CC algorithms. Unlike ALCC, UDT is a UDP-based approach that employs a CC algorithm targeting shared networks.

7 Conclusions

This paper was motivated by the difficulty we face in deploying new CC protocols for cellular networks. Despite
significant advances in CC research for cellular environments, mobile applications are deeply wedded to the legacy TCP stack due to a variety of factors. This paper describes the design and implementation of the ALCC framework to enable migrating CC protocols to the application layer and derive similar performance to the native protocol on top of the legacy TCP stack. We have demonstrated the efficacy of ALCC by integrating three different recent CC protocols (Verus, Copa and Sprout) and showing that the ALCC version of these protocols have very similar performance characteristics (i.e. throughput and delay distributions) achieved by their original versions. We have also demonstrated how easy it is to integrate any CC protocol into ALCC, as well as to incorporate the ALCC framework into existing applications.

References


2 Detailed Response to Reviewer Comments

In the following, we provided a detailed response to the reviewer comments we received in Eurosys 2021.

Comments by Reviewer Review #48A

Thanks for submitting to EuroSys, I enjoyed reading the paper. The subject is relevant, and the solution pragmatic. The paper is didactic and easy to follow. I appreciate that the authors went through the effort of implementing multiple CC on top of ALCC. I particularly appreciate that the proposed solution works on top of TCP-Cubic and can be deployed easily in existing networks.

§5.7 - Figure13: ALCC seems to have a significantly higher latency than Verus. Why is it the case and why wasn’t it the case in §5.2 and Fig11?

Since ALCC runs on top of TPC-Cubic, I would have appreciated knowing the raw throughput and latency numbers of TPC-Cubic in the various configurations. It is currently hard to know if the tested Congestion Control protocols perform better than Cubic -- so it is hard to assess the impact/usefulness of ALCC compared with vanilla TPC-Cubic.

FIGURE 13- §5.7 clarification: In Figure 13, we explicitly introduced packet losses which makes native Verus provide lower throughput while ALCC-Verus recovers from the underlying TCP re-transmissions providing higher throughput albeit with higher delays.

The comparison with vanilla TCP-Cubic is not meaningful since it is shown in previous related work such as Verus and Sprout that TCP-Cubic has an extremely high end-to-end delay in the order of multi-seconds. With ALCC on top of TCP-Cubic, ALCC pace down packets in a more controlled manner, the performance is mainly driven by the ALCC CC protocol rather than the underlying TCP version.

Has ALCC been tested at high speed? 2-5Mbps seems slow for a client (especially over future 5G networks); would the higher CPU usage become a problem at higher speed? On the server side, does ALCC scale with the number of clients or does it introduce bottlenecks (e.g., in the kernel module)?

Please note that in Fig 8c, ALCC is tested for bandwidth settings where the link reaches 40 Mbps (typical in 4G/LTE). We have not tested ALCC in 5G conditions but some recent work has shown that protocols like Verus, Sprout and Copa are known to have performance issues in 5G (which is out of the scope of this paper).
I am not sure I buy the argument against CCP, that it requires kernel modification. It appears that ALCC would also need some way for user-level libraries to be able to read TCP’s congestion window and other state (timers, dupack status, slow start thresholds, etc). Also, like CCP, ALCC is also primarily targeting server-side implementation; so is kernel modification really such a major issue?

We agree that there are libraries for both CCP and ALCC that provide an interface to the underlying TCP stack (the "datapath"). However, in the case of CCP, the user-level implementation consists of two separate programs: a datapath logic program, which dictates the flow and packet level variables that should be reported to the congestion control algorithm, and then the congestion control algorithm itself. By "datapath" in the context of CCP, this is what we mean. In contrast, ALCC does not necessarily require a Kernel module, and it can also run without relying on any TCP information. This is done by implementing ALCC’s own acknowledgment mechanism, given that even in the server-side only implementation the only information that ALCC required was the TCP acknowledgments. We have added a new section on integrating ALCC within mobile phones that does not require any Kernel support or modification to the underlying transport layer stack (see section 4.3).

What do you anticipate using for streaming data out of a phone, as opposed to into it? That data transmissions would also be subject to issues like bufferbloat, and unpredictable performance. Wouldn’t ALCC make sense there too? I guess the issue of high CPU overhead is much more significant there (more on this below).

This is a very valid point, and yes ALCC does make perfect sense here. For uplink scenarios like the above one, ALCC can be implemented at the mobile phone. We have added a discussion on how this can be done in section 4.3.

CPU overhead of ALCC is a bit bothersome; it appears to nearly double the utilization. While this may be fine for server-side deployments, it is definitely an issue for mobile deployments. Even for server-side the trend is to move computation away from CPUs and into hardware, so the approach that ALCC is advocating is somewhat anachronistic. It would be good to have a discussion on the trade-offs of ALCC’s approach vis a vis recent hardware-assisted congestion control schemes. Also, the authors may want to look at recent work on light weight packet scheduling (using timing wheels) as this may help mitigate some of the CPU overhead issues you are facing. One the whole, it was a little disappointment that not much effort was devoted to being CPU efficient.

CPU overhead numbers are comparable to CCP, and our implementation can be optimized further to support a larger number of clients. We have added a detailed discussion in the updated version. Section 5.6 highlights the CPU utilization. We have added a clear discussion in Section 4.3 about
implementing ALCC on a mobile device, which includes using a lightweight alternative for CC, concerning the CPU overhead in mobile deployments. Also given that there is no need for using a Kernel module in the mobile scenario, this will reduce the CPU utilization significantly.

Are there any situations where the ALCC’s application level logic may make decisions inconsistent with the underlying congestion control protocols? This is a broader question that merits deeper discussion, as it relates to the composability of congestion control algorithms at a fundamental level: while using two cwnds is a simple way to compose, congestion control algorithms differ vastly in their fine time-scale dynamics; TCP’s dynamic control loop is different from recent cellular protocol proposals’ loops, for instance. It is not immediately obvious how to compose such control loops without creating some stability issues. A more principled discussion of this, in addition to providing a framework that allows composability, would have been a lot nicer.

We have added a detailed discussion on this issue. In cellular networks, Cubic typically reaches a high window during slow-start that this scenario normally doesn’t arise often for Verus, Sprout and Copa at the ALCC layer since they operate in lower window, lower delay regimes. However, when the ALCC window is bigger, the underlying TCP socket blocks ALCC from sending more data, thereby restricting the ALCC transmission rate which in turn may be used by the ALCC delay based protocol for adapting its sending rate.

Comments by Reviewer Review #48C

ALCC is a framework to move TCP congestion control into the application layer with minimal modification to the TCP stack. The core idea is that the application can, in effect, enforce a smaller congestion window by passing less data to the kernel to send. Combining this with the fact that in the mobile space the TCP congestion window with standard algorithms is typically much too large, allows congestion control to be moved into the application without modifying the kernel TCP stack (significantly).

It took me a really long time (part way through sec 4 at least) till I finally understood what this paper is doing. Here are some of those confusing statements:

"ALCC leverages the kernel TCP implementation without the additional complexity of directly modifying the data." (modifying what data?)

By data we mean the datapath. Thank You for highlighting this, we have now corrected this typo.

"at the application layer without modifying the underlying transport layer" (the transport layer protocol or the implementation?)

We are neither modifying the transport layer protocol nor it’s implementation.
"without any modifications to TCP" (protocol or implementation?)

Without any modifications to either.

"implemented in the userspace as a wrapper around the default Linux BSD TCP implementation" (presumably this is referring to the sockets API?)

That is correct.

Overall the description just made it really hard to figure out what you're actually changing and what you're looking on keeping the same. This can be described much more crisply. The core idea is so simple, there is the gotcha of dealing with ACKs and that's mostly it. I think the high-level of all of this can be described in the intro already.

We added a higher level description of the ALCC idea in the intro.

Also intro should make clear that there are other things such as CCP and QUIC that have similar goals but implement them differently. I was constantly thinking "what about CCP? isn't this CCP?" until I finally got to that comparison later on.

We have added this in intro mentioning that they are other frameworks with similar goals.

The comparisons to QUIC and especially CCP felt half-hearted and defensive. A lot of the discussion focuses on implementation details like the lisp syntax, the rust implementation of CCP, etc. And clearly QUIC is being used broadly, so if there is problems in the mobile space that cause it to not be workable those are presumably well known and can be cited and that will be that for the comparison. The fact that the current implementation seems to make it tricky to implement other CC algorithms is really unconvincing. Focus on fundamental differences.

We have tried to adjust the writing and focused on more fundamental differences. We have even reached out to the CCP authors and consulted them in discussing the differences between the two frameworks.

The paper describes two implementations, one-sided with kernel integration for ACKs, or two-sided without kernel modification but with application layer ACKs. This is interesting. But the paper does not talk much about what the differences are, when which one will be useful etc. Again this should come early on. Just throwing out options is unconvincing. E.g. is just running on the server what you expect to be the common case? When would the client-server version be useful?
We have added a section on mobile phone integration of ALCC, which is where the two-sided implementation is crucial. The single-sided implementation is mainly when the target application is in the downlink direction. This makes the integration super easy, given that there is no need to change anything on the mobile phone side. However, in the uplink direction, it is hard to do a Kernel implementation on Android. It is required that ALCC has its own acknowledgment mechanism, hence the two-sided implementation.

Finally, the evaluation felt a bit shallow. Repetition of three different algorithms all with similar results comparing to baseline. But the actually interesting bits, like where the differences are coming from in there, are mostly ignored or just brushed over. Those differences are what is truly interesting, besides it mostly just working good enough. Specifically:

- 5.2: Unfortunately a one sentence description of "This minor penalty arises from running a second control loop on top of the underlying TCP." does not provide much information.
- 5.4: does no talk about where the differences come from at all.

We have expanded the explanation on both of the above suggested points.

In the motivation, the paper talks about challenges of split TCP connections. But the later part of the paper never circles back around to those. Does ALCC help here? Is there interesting properties if you enforce CC at the application layer with the client-server version where the ACKs actually come from the receiver? This seems really interesting and there could be something fundamental here that would make the story much more compelling.

We added details on the split TCP and also discussed how ALCC can help overcome some of these challenges when implementing ALCC for uplink on mobile phones (section 4.3).

Smaller suggestions and nits:

Some of the text (especially in 2-4) is at times a bit repetitive and similar statements come at the beginning of each section and sometimes subsection. While I appreciate some sign-posting, this really stuck out here as repetitive

We tried to cut down the repetitive parts.

Applying the claim "none of these solutions have gained wide deployment over real network conditions" to QUIC seems like more than a stretch. From what I understand Chrome ships with QUIC support and google defaults to it, Facebook also relies on QUIC, and android has native support.

Thank you for highlighting this. This was a mistake, we were basically referring to CC protocols and not QUIC or SPDY frameworks. We have removed the citations from this paragraph.
similarly: "[Quic] does not allow for integrating into other applications apart from some Google services such as YouTube or Google search." seems wrong.

We have removed this as well.

pull up table 1 to prev. page

Moved up.

4.1.3, 4.3 went on for too long, lots of uninteresting implementation details. Does really not add anything, space could be better used for other things.

We believe this is an important description to highlight the simplicity of integrating existing CC protocols in the ALCC framework.

Somewhat unrelated:

But are there any additional benefits to running the CC-loop inside the application? Like application-layer integration with CC, prioritizing of RPCs or something like that? That might make this more convincing.

The fact that the CC protocol can be moved even above the application layer protocols (such as HTTP) gives a huge benefit. Especially when it comes to tunnels and middle-boxes. Where ALCC does maintain an end-to-end CC loop between the two end-hosts.

Comments by Reviewer Review #48D

The paper observes that despite all the newly proposed congestion-control (CC) protocols for cellular networks which suffer the know buffer bloat problem (over provisioning the network buffers to avoid packet losses) which leads to long end-to-end packet delays, mobile apps are still tied to the legacyTCP stack due to many reasons, and proposes the ALCC framework that essentially runs new CC protocols at the application layer on top of legacy TCP stack running in the kernel.

The paper focuses on the implementation details of such a framework, how easy it is to port new CC protocols into the framework, and evaluation of 3 contemporary CC protocols in the framework. Results show all CC protocols ported the framework can achieve very similar throughput and packet delay as their native (kernel) implementations. The paper gave a page or so intuition of why ALCC would work. The key insight is that when running in cellular networks, the TCP inside the kernel will have unnecessarily high congestion windows, as long as the application-level CC congestion window is no larger than the otherwise kernel CC congestion window, the application level CC should be effective.

I enjoyed reading the paper and liked the idea and can be believe why the idea (running an application level CC on top of a legacy TCP) would work well, but I wish there were some theoretical
analysis to explain why the coupled system would always work well and will not suffer adverse interactions of the two layered congestion control protocols.

Short of which, I would have liked to see some in-depth microscopic analysis of one CC protocol e.g. Verus operating on top TCP (say Cubic) version running natively to understand why its micro-level behavior is not affected by the CC of the TCP below.

The native TCP behavior has been widely studied in papers such as Verus and Sprout. The performance of native TCP protocols such as Cubic is well known to have a widely large congestion window and result in multi-second end-to-end delays. In this paper, we focused on showing how a CC protocol can operate on top of existing TCP versions such as Cubic, while retaining the same behavior of the native CC protocol without the multi-second end-to-end delays.

Short of above, the paper feels like a bit magic showing yes it works but without deeper (theoretical or anything else) explanation as to why.

It did feel like magic for us too at the beginning :-). However, the idea is very simple. Leveraging the fact that Cubic for example opens a huge congestion window, by sending data at a much lower rate than the unnecessarily large TCP window, ALCC can trick TCP and achieve the desired CC protocol benefits. Its simply utilizing the big fat pipe of TCP and sending lower than that is what makes the whole thing works.

1.5x - 2X CPU overhead is not trivial. What about memory overhead, given ALCC maintains its own queue

Unfortunately, we have not tested for the memory overhead. However, we believe it will be similar to the native CC protocols used within ALCC, given that even the native versions of these CC protocols would have to do exactly the same buffering at the application layer.

The operational region plots only include 25% -- 75% percentile data points. First it is not clear is this for points that fall in 25%-75% percentile in terms of both throughput AND latency, or throughput OR latency? If it is the former, then the portion of points included can be really small (worst case only 1/4?). Even if it is about 50% of all the measurements, it is still rather small. What will the figure look if you include more measurement points?

This was inspired by the box-plot, where it shows the 25% - 75% percentile numbers of the overall data from the distribution.

The x-axis unit in all Operational regions figures says s (second), did you mean ms?

Thanks for highlighting this, and yes you are correct this was a typo and it is supposed to be ms. We have changed this.
Comments by Reviewer Review #48E

This paper proposes ALCC in order to evolve cellular networks’ congestion control by layering it in userspace on top of the existing, and deeply entrenched, TCP stack. The paper applies ALCC to Verus, Copa, and Sprout and illustrates that adapting these CC proposals to ALCC requires little effort. Retaining the traditional TCP socket interface also makes it easy to port applications to use ALCC. Finally, the paper presents a trace-based evaluation on top of the Mahimahi emulator that shows that ALCC variants of the three cellular congestion controllers have similar delay/throughput characteristics to native controllers, that are not layered on top of TCP.

The problem and proposed solution in this paper is great: build congestion controllers on top of ALCC and support them from the user-space. This can be used, as the paper demonstrates, to deploy new/novel controllers without changing the underlying, particularly client-side, TCP stack. In the cellular networking context this is a big win.

On the research side, however, I see little novelty over CCP [26]. For example, ALCC assumes a particular datapath (kernel TCP stack), which means that it is a special case of CCP. CCP supports QUIC, mTCP and DPDK datapaths.

While both ALCC and CCP migrate congestion control (CC) to the application layer, ALCC addresses a fundamentally different problem. ALCC is specifically designed to support delay-based protocols to operate efficiently at the application layer for mobile applications and services on top of underlying TCP tunnels in cellular networks. In cellular networks, TCP tunnels are very commonly used for: (i) mobile devices that are behind a NAT/Proxy; (ii) Evolved Packet Core (3GPP) for supporting cellular networking functions; (iii) Third party applications/services built for using secure tunnels to middleboxes. Mobile applications/services have NO control/visibility over the underlying tunnel and are exposed to the standard send/recv interface. And the problem with TCP in cellular networks is that the buffers are large, resulting in large packet delays, and ALCC smartly addresses this problem by controlling TCP windows from the application layer. In such settings, a generic framework like CCP isn’t really beneficial because of the effort involved in implementing a new congestion control algorithm on a mobile device, even if it is at the application layer.

I read section 2.1.1 carefully to understand the comparison between ALCC and CCP. The argument for why ALCC is superior seems to be based around (1) programmability and (2) deployability: CCP is harder to use than ALCC because of its different interface, and ALCC exposes a familiar socket-style interface to application. Both of these are not convincing to me. For (1) there is no mention of the fact that CCP is more general, which means that ALCC cannot implement everything that is expressible with CCP. There is also no evaluation of ‘difficulty’ from the implementer side. For (2), as far as I can tell, CCP does _not_ change the datapath interface that applications use -- applications that use TCP sockets continue to use them unchanged with CCP. By contrast, ALCC requires updating applications to use ALCC-style sockets.

We tried to expand the description on how different ALCC is to CCP. We have even consulted the CCP authors and their feedback on the difference was: “There is a tradeoff space here, which I
think could be emphasized more - CCP requires datapath modifications and a new CC API, while ALCC requires no modifications to anything, and presumably your API is more similar to traditional per-packet implementations. From my reading, ALCC is basically saying, if I am in this specific cellular environment where the datapath is set in stone, you are offering an easier path to deployment than waiting for datapath features.”

Considering how similar the ALCC idea is to CCP, I was surprised that there is no direct head-to-head evaluation of the two frameworks. CCP is open source and even includes implementations for some of the same congestion controllers that this paper evaluates, such as Copa. This is a major omission.

Given that ALCC is trying to solve a different problem than the CCP, we did not really compare both frameworks head to head even when specific shared protocols such as Copa were used. We experimented a bit with CCP Copa, but many of the tuning parameters were not present in the CCP implementation of Copa, making the comparison unfair.

I think the paper can be strengthened by demonstrating that CCP is not well suited to the cellular network context and that this domain requires a different approach, namely ALCC. I think that this demonstration needs to be done experimentally -- you need to show that either CCP cannot do what ALCC does, or CCP cannot provide the same capability without some fundamental trade-off, such as lower performance.

This is an excellent suggestion; we have tried to explain why CCP fails in certain aspects within the cellular scenarios, especially when it comes to the uplink.

I’m not familiar with Mahimahi and its limitations; these would be useful to discuss. I was also slightly taken aback by a pure trace-base emulation for an evaluation. You have an implementation, so why not run it and include experimental results from a deployment?

The problem with real deployments is reproducibility. There are many random factors, such as different channel characteristics or different background loads, that are not under our control when running these protocols over real cellular networks. And as a result, comparing different protocols against each other might not be fair given that there might be random events happening at the cellular base station that affect a particular run’s performance. Mahimahi can control these factors and run these protocols with the same channel characteristics, making the comparisons fairer and can be attributed to the different frameworks and CC protocols.

**Comments by Reviewer Review #48F**

ALCC is an application-level congestion control (CC) framework. The authors motivate for ALCC as follows. First, CC in legacy TCP stack unnecessarily reduces throughput because it treats packet
losses in cellular networks – common events due to user mobility or cellular network properties – as congestions. Second, network operators cope with this issue by over-provision their network buffers, leading to large packet delays. Third, while existing research proposed new CC protocols for cellular networks, they either rely on UDP, at odds with the TCP-based middleboxes or require tricky re-implementations of CC protocols. ALCC solves the TCP-vs-UDP problem by doing application-level CC on top of TCP. It provides an interface similar to Berkeley socket API with alcc_accept, alcc_send, alcc_close and the authors argue that this familiar interface eases implementing or porting CC protocols. The authors built two ALCC implementations, a server-only implementation which requires a kernel module to expose TCP ack info and a client-server implementation purely at the application level. Evaluation using a network emulator and traces shows that ALCC with CC protocols Verus, Sprout, and Copa behave similarly as their native implementations.

Thank you for submitting to EuroSys! I enjoyed reading your paper and learning how ALCC works. I, too, would like my mobile apps to enjoy low network latency and high bandwidth, and I applaud your effort for doing research towards this goal. I have two suggestions to further improve your paper.

First, please spell out the fine design points and assumptions of ALCC so that the work becomes more convincing. As you explained, your server-only ALCC implementation cannot support Sprout so you had to build a client-server implementation. How do we know that your current design suffices to support new CC protocols and requires no additional big changes? The interface that ALCC exposes seems to include alcc_accept, alcc_send, alcc_close, and some methods to get ack info, but the paper does not seem to describe this interface clearly at one place. It is thus hard to understand why this interface suffices to support many CC protocols. ALCC runs on top of an existing TCP implementation, and as you explained, if the underlying TCP uses a large CC window, ALCC can restrict this window at the application level. Is it possible that the underlying TCP window is small but ALCC desires a larger one?

Only a small number of socket library interfaces are provided/exposed to the application layer. We have mainly chosen the above three due to their effect on the framework’s various designs’ impact. In general, CC protocols rely on three different components: packet inter-arrival time, acknowledgments, and losses. For Sprout, given that the CC protocol needs to send extra data back from the client, we had to amend the implementation with our own acknowledgment mechanism. Hence any similar CC protocol that requires sending additional info can reply on this mechanism. We can also use this when integrating ALCC into the Android mobile OS stack.

For the congestion window, if TCP has a smaller window, it directly reflects the channel condition. Even if ALCC desires a larger window, it will notice the TCP socket blocking signal indicating congestion in the network, and ALCC needs to slow down.
Second, please consider strengthening your evaluation to show the ease of building new CC protocols with ALCC, as this is the key selling point of your work compared to prior application-level CC frameworks CCP and QUIC. It appears that only three CC protocols are implemented with ALCC. Is it possible to include many more? Also, could you please quantify the effort implementing CC protocols in ALCC vs in CCP and QUIC? For starters, perhaps count the lines of code written and show a CC protocol example in ALCC. You may also consider a user study if possible.

Section 4.1.3 discusses in-depth the implementation details of integrating CC protocols into ALCC. We discuss how we integrated two CC protocols, Verus and Copa, into ALCC. We did not show the number of lines needed for the integration since ALCC reuses the same code of the CC protocol but splits it across the multiple parts of the ALCC framework.

Writing can be dramatically improved. Consider reorganizing the description of ALCC to present the complete interface of ALCC and example implementations of CC in ALCC. The text is verbose at many places. Section 1 and 2 take almost three pages but they contain much redundancy. Please rewrite to make them more concise.

This was taken into consideration when we re-edited the current paper version.

Minor comments:

- Lines 81-98. They can probably be summarized into a few sentences to make the main points more crisp.
- Lines 291-294. Please include citations to these sources and possibly some direct quotes to make the point more convincing.
- Lines 402-403. "... we implemented [the] shim layer to randomly [chose]..."
- Lines 802-805. Why can't this loop be inside alcc_send()?

Addressed.

- Figure 6. I don't understand these figures. Why are these shapes so regular?

These were inspired by the box-plots across two different axes.

Comments by Reviewer Comment @A1 by Reviewer B

This paper was briefly discussed online. The reviewers were unanimous that the paper presents an interesting idea, but that it needs to better position the work relative to CCP, as many key ideas
seem similar. A quantitative comparison would be best. Reviewers also felt that this issue, along with the issue of congestion control interaction, was not well addressed by the paper or the rebuttal.