Supercharge WebRTC: Accelerate TURN Services with eBPF/XDP

Tamás Lévai
Budapest University of Technology and Economics
L7mp Technologies
levait@tmit.bme.hu

Balázs Edvárd Kreith
Riverside.fm
balazs.kreith@gmail.com

Gábor Rétvári
Budapest University of Technology and Economics
L7mp Technologies
retvari@tmit.bme.hu

ABSTRACT

Real-time communication (RTC) services, from videoconferencing to cloud gaming and remote rendering, are everywhere. WebRTC, an enabler technology for these applications, traditionally relies on a comprehensive NAT traversal protocol suite, most importantly, TURN, to interconnect clients and media servers behind NATs and firewalls. With the demise of residential public IP addresses, these massive-scale TURN services have become an indispensable component of WebRTC applications. Traditionally implemented as multi-protocol user-space packet relays, TURN servers are notoriously resource hungry. In this paper, we propose an eBPF/XDP offload engine to improve TURN server performance. We design a reusable eBPF/XDP TURN offload architecture, create a prototype on top of `pion/turn`, a popular WebRTC framework written in Go, and show on a fully functional WebRTC testbed that our offload significantly improves throughput and, more importantly, delay, by 2–3× compared to the state-of-the-art.

CCS CONCEPTS

• **Information systems** → **Multimedia information systems**;

KEYWORDS

TURN protocol, WebRTC, eBPF/XDP

ACM Reference Format:
Tamás Lévai, Balázs Edvárd Kreith, and Gábor Rétvári. 2023. Supercharge WebRTC: Accelerate TURN Services with eBPF/XDP. In *1st Workshop on eBPF and Kernel Extensions (eBPF '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609021.3609296

1 INTRODUCTION

Today videoconferencing (Google Meet, MS Teams, Discord) and cloud gaming (GeForce NOW) are part of our everyday life. Most such *real-time communication* (RTC) services rely on WebRTC [1], an extensive API for handling real-time media (e.g., audio, video) in Web browsers, plus a bunch of legacy protocols, for the most part adopted from Voice over IP (VoIP), for transmitting media over the Internet.

eBPF '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0293-8/23/09.
https://doi.org/10.1145/3609021.3609296

Increasingly, users connect to WebRTC services from behind one or more layers of network address translators (NAT). As an unfortunate legacy of VoIP, however, WebRTC media communication requires direct connection between peers. In response, a comprehensive suite of NAT-traversal protocols have been standardized in the IETF, and implemented in browsers and server-side WebRTC frameworks, to help clients traverse NATs. Clients and servers negotiate the connection parameters using ICE (Interactive Connectivity Establishment), exchanging a list of candidate IP address and UDP/TCP port pairs that may potentially allow them connect. ICE candidates in turn are obtained via an additional set of protocols; for instance, STUN (Session Traversal Utilities for NAT) allows peers to "punch a hole" at the outermost NAT layer, which, in certain lucky situations, may allow them to connect directly. When all attempts fail, WebRTC clients and servers fall back to TURN (Traversal Using Relays around NAT, [16]) to relay traffic via a public proxy server. TURN services also find extensive use beyond WebRTC, like VPNs [4], desktop streaming, game servers, etc.

The typical *standalone* TURN service runs on one or more dedicated servers over a public IP address. A recent alternative is to *bundle* the TURN server into the WebRTC media server [21] (e.g., LiveSwitch, Web Call Server); this cuts down the round-trip latency from clients via the TURN server to the media servers. In addition, TURN is increasingly being adopted as a *media gateway* protocol to ingest media traffic into a Kubernetes cluster, allowing to run media servers on private IPs inside the cluster (STUNner, [8]).

Relaying high-definition live video and audio streams for potentially millions of users in real-time, TURN servers are notoriously resource and network intensive. Hosting a public TURN service requires significant expertise, vast bandwidth, and high-end server CPUs. Worse yet, most open-source TURN server implementations run in user space, processing massive-scale traffic comprising predominantly small UDP packets, which is a well-known performance corner case for the Linux network stack [3]. This calls for new network optimization techniques. Unfortunately, rewriting the TURN server over a high-performance user-space network stack, like DPDK, would require prohibitive effort and hardly fit into the modern Kubernetes ecosystem [8].

In this paper we propose an alternative approach using eBPF/XDP as a portable TURN offload. Our contributions are as follows:
eBPF/XDP acceleration for TURN. After introducing the main mechanisms in TURN (§2), we present an effective design for accelerating TURN relays by offloading packet processing to an XDP program (§3).

Integration. We present a practical implementation of our design on top of the popular `pion/turn` library (§4).
Evaluation. We conduct extensive evaluations to understand TURN performance in microbenchmarks as well as in end-to-end meaurements (§5).

We close the paper discussing related work (§6) and deriving the main conclusions (§7). We plan to publish the offload engine under a permissive open-sourse license and contribute it back to the upstream `pion/turn` library.
This work does not raise any ethical issues.

2 TURN IN WEBRTC

TURN (Traversal Using Relays around NAT, [16]) is an IETF protocol for connecting WebRTC clients and servers via a public relay when direct communication is not feasible due to a restrictive NAT in the media path. With the depletion of the public IPv4 address pool and the trend towards hosting end-users behind carrier-grade NATs, TURN has become a necessity today [9].

A TURN session starts by a client (e.g., end-user) connecting to the TURN server on a well-known port and sending an Allocate Request to it. TURN supports essentially any possible transport protocol (e.g., UDP, TCP, TLS/TCP, DTLS/UDP), but most applications prefer UDP to minimize latency. Once the client is authenticated, the TURN server creates an *allocation* opening a so called *transport relay connection* on behalf of the client, and sends the IP address and port of this connection to the client. The client can then send this address and port to its peers and the TURN server makes sure to relay all packets between the client and the peers connected to the transport relay connection. Any number of peers can connect; this makes it possible to host multiparty conferences over TURN. Since the transport relay connection is typically created on a public IP address (although this is not strictly necessary [8]), the connection is guaranteed to succeed even when all peers are behind a NAT. Clients are uniquely identified by the IP 5-tuple (client IP and port, TURN server IP/port), and similarly for peers.

If a client wishes to send a packet to a peer, it encapsulates the payload into a Send Indication message, prefixes the payload with the IP address of the peer, and sends the message along to the TURN server. The server will decapsulate the message and relay the payload to the intended peer over the transport relay connection. Due to the additional TURN header, this communication mode comes with large overhead, which adds significantly to the otherwise already costly TURN server bandwidth fees. To cut down the overhead, TURN supports a lightweight communication mode called *channels*. A TURN channel is essentially a direct shortcut between the client and a peer via the TURN server. Channels need to be explicitly created with a Channel Request message and each channel is identified by a 2-byte channel id that is unique within the allocation. Then, the client can send any data to the peer inside a ChannelData message, which is essentially a shim 4-byte prefix before the payload that contains the channel id. The server parses the channel id to look up the corresponding peer, removes the channel id, and sends the packet on to the peer. Communication in the reverse direction occurs similarly: the peer sends a plain UDP/TCP packet over the transport relay connection, the server looks up the corresponding channel id, encapsulates the payload prefixed by the id, and sends the resultant ChannelData packet back to the client. Channels remain active as long as clients actively refresh them.

Thanks to the small overhead, most TURN clients default to sending all data over channels (instead of Send Indications). Accordingly, the bulk of the TURN server workload (more than 96% of TURN packets as reported by our measurement with a Chrome client and a LiveKit server) involves relaying Channel Data packets from clients to peers and vice versa.

Implementing high-performance TURN servers is tricky. Typical TURN traffic consists of lots of small UDP packets that need to be processed at very high speed (a single 4K video consumes 32Mbps!) and at extremely low latency. This workload is known to pose performance regressions due to the frequent copying of packet data back and forth between the Linux kernel and the user space TURN server [3]. For TURN server implementations written in a managed language, frequent (per-packet) memory allocations add substantial extra overhead [13]. For instance, the `pion/turn` library is written in Go, which, thanks to its versatility and extensibility, has made it one of the most popular open-source TURN implementations. Unfortunately, `pion/turn` in UDP mode maxes out at about 30-50 kpps (thousand packets per second) and introduces several milliseconds latency and jitter. We have recently contributed several performance optimizations to the `pion/turn` project, including a full multi-threaded UDP/TURN listener implementation [19]. Still, our benchmarks indicate frequent performance regressions (see §5). In this paper, we introduce a new eBPF/XDP acceleration framework to remedy this situation.

3 ACCELERATING TURN CHANNELS WITH XDP

3.1 Architecture

Rather than implementing a complete TURN server in kernel, we adopt the fast-path–slow-path separation principle [12]. We propose an eBPF offload architecture that enables processing the bulk of the TURN server workload in the kernel at very high speed and small latency, and let the "difficult" packets, which constitute only a tiny fraction of the typical TURN traffic mix, be handled by the conventional user-space TURN server. In this paper we present the XDP offload, but the architecture is easy to extend to support additional eBPF offload techniques (i.e., eBPF/tc, sockmap).

We focus on accelerating the relaying operation for ChannelData messages, the hotspot in a TURN server due to their sheer number. It is also simpler to implement channel data processing in the kernel as opposed to managing allocations and permissions. As Fig. 1 shows, we keep most of the TURN server functionality in user space, and offload only ChannelData processing to the XDP program running in the kernel. The XDP program and the user space communicate via eBPF maps.

The XDP program processes only ChannelData messages and passes the rest of the TURN messages to the server running in user space. If a new client connects to the server, the connection establishment (allocate request, channel binding) will be handled in user space. At the end of a successful channel binding, the user-space server registers the new channel in the XDP program and from that point the bulk of client traffic (channel data) is handled by the XDP program in an efficient way, while periodic refresh

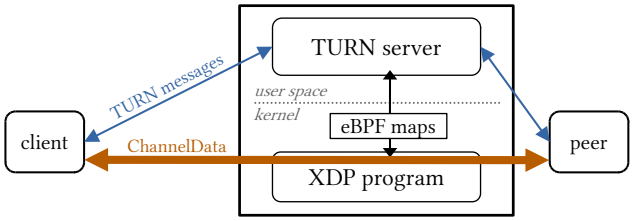

Figure 1: Architecture: After a successful channel binding, media packets take the kernel fast path. Other TURN messages are processed by the user-space server.

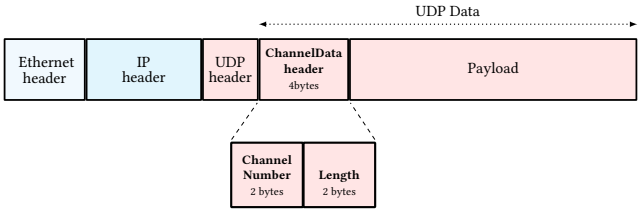

Figure 2: ChannelData Packet Diagram.

messages are passed on to the server. When the client disconnects the user space unregisters the channel from the XDP program.

Next, we detail the main challenges we found in applying an XDP offload for ChannelData TURN messages.

3.2 Resizing UDP Packets

As Fig. 2 shows, the ChannelData header starts at the L4 protocol payload (e.g., UDP Data), and is 4-byte long with two fields: a Channel Number (2 bytes) to identify the channel, and Length (2 bytes) to specify the length of application data that follows. The TURN server inserts the header when it relays peer data to the client, or removes it in the reverse direction.

This step is challenging to support in XDP. Incoming packets are represented as a packet buffer that is a continuous memory area. Adding or removing bytes is supported only at the beginning and the end of that packet buffer. Consequently, to add/remove a 4-byte field at the beginning of the UDP payload that is in the middle of the buffer, the subsequent chunk of data needs to be shifted by 4 bytes. Moving variable size packet payload in memory, however, is difficult in eBPF because `memmove` is limited to a small constant size.

A straightforward solution is to call `memmove` in a loop. Shifting by 4 bytes at each iteration results large number of loop-cycles even in case of small packets, and limits the size of packets that can be processed. An alternative option to shift larger chunks whenever possible; in other words, to shift in batches. Shifting a single 32-byte chunk can replace eight 4-byte shifts as shown in Fig. 3. Unfortunately, this is still difficult to implement because the eBPF verifier prevents loops that cannot be proven to be bounded.

An alternative appoach would be to unroll the `memmove` loop. Unfortunately, as the size of the UDP payload is variable, we do not know the amount of data that needs to be moved, only the maximum payload size. The compiler in this case generates loops

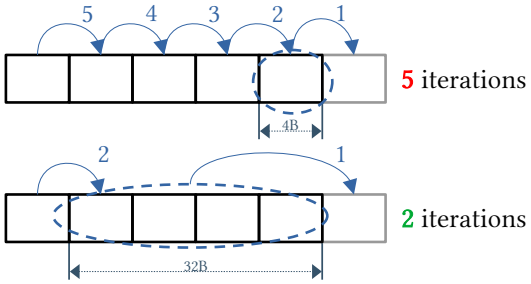

Figure 3: Shifting packet data to insert ChannelData header: `memmove` large chunks minimizes loops needed.

considering the maximum size (e.g., 1480 bytes), resulting a large program. The limited size of eBPF programs can become a hard limit on the payload length we can support with this approach.

Instead, we settled with an option that minimizes the number of loop iterations. Our solution is based on a simple observation: it is easier to shift the small constant size data chunk that comes *before* the channel id with a negative offset than it is to move the variable size chunk that comes *after* it with a positive offset. This first chunk consists of the L2, L3 and L4 headers and is of fixed size for the typical Internet packet (e.g., 46 bytes for Ethernet+IPv4+UDP and 66 bytes for Ethernet+IPv6+UDP), which is much smaller than the variable size payload. Once the XDP program parses the packet headers it knows *exactly* how many bytes to move. Our code contains specialized data shift implementations for the most frequent header combinations and it is easy to add more.[1] Packets that contain an exotic header (e.g., an IP option) are passed on to the slow path.

3.3 UDP Checksum Calculation

While relaying a ChannelData message, the underlying UDP pseudoheader and data gets modified. This renders the UDP checksum invalid, and might result packet drop on the receiver's end. UDP checksum originally is a 16-bit one's complement of the one's complement sum of the pseudoheader and the UDP packet (plus padding if needed) [14].

XDP programs handle UDP header updates (e.g., changes of source/destination port) vi an efficient incremental checksum calculation method [17] that calculates a differential checksum update based on the prior state and the current state of a modified chunk of the packet. Adding or removing the ChannelData header to UDP data is not suitable for this differential checksum update algorithm due to the lack of prior state. A naïve approach is to either recalculate the full checksum, which is computation-intensive and has an upper limit on packet length due to loops in checksum calculation; or to generate no UDP checksum, which works for IPv4 only.

Instead, we use an efficient incremental checksum update algorithm. This method relies on the commutativity of one's complement sum [18]; adding 4 bytes to the beginning of the data has the same effect on the checksum as adding it to arbitrary position. In addition, adding 4 bytes does not require additional padding for

[1] Cilium follows a similar approach: https://github.com/cilium/cilium/blob/b6235e4a7f97c783d928f89b1c0f8a89a52d83c7/bpf/include/bpf/ctx/xdp.h#L270

the calculation. The implementation relies on the `bpf_csum_diff` helper function. A similar approach works in the reverse direction, when the program removes the ChannelData header.

3.4 Monitoring

Monitoring the way media traffic is processed through the TURN server, which is inserted at a particularly sensitive point into the real-time media plane, is crucial. Most TURN server implementations therefore contain broad support for generating monitoring data that can be visualized, for instance, in Prometheus and Grafana. This lets operators to quickly spot the origin of performance hotspots and latency spikes.

When network traffic does not flow directly through the TURN server, for instance because we shortcut channels in the XDP offload, these monitoring functions become ineffective. To overcome this problem, our XDP program provides the instantaneous packet statistics in an eBPF map (number of packets, bytes, etc., by active IP 5-tuple). Conversion to connections, statistics aggregation, and Prometheus integration is still delegated to the user-space TURN server.

4 IMPLEMENTATION

We implemented the XDP acceleration for TURN channels for client–peer and peer–client connections using the `cilium/ebpf` package and integrated the offload with the `pion/turn` library. The `cilium/ebpf` library is a Go tool for building eBPF-based applications, supporting an extensive range of hooks to attach programs. The `pion/turn` library is an easy-to-extend TURN server implementation written in Go. We added the XDP-acceleration as a reusable Go package of `pion/turn`. Our implementation handles only UDP transport at the moment; channels using different transport protocols are handled on the slow path. The XDP program is roughly 300 lines of C code, and the `pion/turn` integration took appr. 350 lines of Go code. The source code is available on GitHub [10], we are also working on upstreaming our changes.

5 EVALUATION

In this Section, we evaluate the performance aspects of applying eBPF/XDP offload for a TURN server to process TURN ChannelData messages. During the evaluations, we focus on two aspects: *i)* investigate internal TURN server performance improvement, and *ii)* get insights into the end-to-end performance of the XDP-offloaded TURN server.

5.1 Evaluation Setup

We use two baseline TURN servers: the single-threaded `pion/turn` example server implementation (this is the default operational mode of `pion/turn`), and a multi-threaded `pion/turn` example server that allocates 4 threads to handle UDP sockets. The source code is available on GitHub [10].

Our testbed consists of two servers connected back-to-back with a full-duplex 40G link. Both servers are equipped with 12×2.4GHz CPU (power-saving disabled) and 64GB RAM installed with Ubuntu 22.04 (kernel: v5.15.98).

5.2 Microbenchmarks

The first set of measurements focuses on quantifying the TURN server performance gain by offloading its packet processing logic to the kernel partially. In these measurements we try to eliminate external factors (e.g., networking). For this purpose, we evaluate the offload using `pion/turn` benchmarking and identify the key speedup point in its execution. In addition, we present an end-to-end microbenchmark on the loopback interface.

Standalone benchmarks. Our first performance evaluation relies on the official `pion/turn` benchmarking capabilities. The benchmark instantiates and interconnects a minimal TURN server, a client, and a peer, on the localhost. The client transmits data to the peer via the server for a given time period, then terminates the connection. During the execution, Go profiling collects execution data. We examine this profiling data, and draw conclusions.

We execute the benchmarks with and without the ChannelData XDP offloading. The profiling data involves the client and peer packet processing, as well as the server. The server's packet processing hotspot is the `Server.readLoop()` function that manages incoming traffic; it classifies TURN messages and reacts to them (e.g., does the ChannelData processing).

Without the offload, the `Server.readLoop()` takes a significant part of the execution: 47.89s (Fig. 4). This is roughly 34% of the total running time of the benchmark; the rest is comprised by the clients generating test traffic and a sink receiving it. The benchmarks indicate that our offload essentially removes this processing time altogether: with XDP offload the total running time of `Server.readLoop()` is now only 0.96s, or roughly 0.1% of the total running time (Fig. 5). This is more than 50× improvement. Consequently, the overall execution time of the benchmark decreases from 113.76s to 63.14s. These results motivate the application of eBPF/XDP offloading in TURN servers.

Testing with real traffic. In our next measurement, we benchmark the TURN server implementations in a more realistic environment. We create an evaluation setup (Fig. 6), in which we have a fine-control over the test traffic. First, we use the loopback interface to connect the components; later, we show end-to-end results.

We execute 3 consecutive 100-sec measurements and summarize the steady-state results in Table 1. The single-threaded default `pion/turn` implementation yields limited performance. Increasing the number of processing threads seems to improve on this baseline significantly: with 4 threads we see an 3× throughput improvement with a slight increase in average delay. The XDP offload improves the throughput by 6.4× compared to the single-threaded case. Interestingly, we found that the main bottleneck is not the TURN server any more; rather it is now the `iperf` server that constrains throughput. A quick baseline measurement between an `iperf` client and server yields a hard 230 kpps limit, which indicates that there is still significant room for loading our XDP accelerated TURN server further. In addition, the XDP offload improves delays by 10×. For real-time communication services, such performance improvements are crucial.

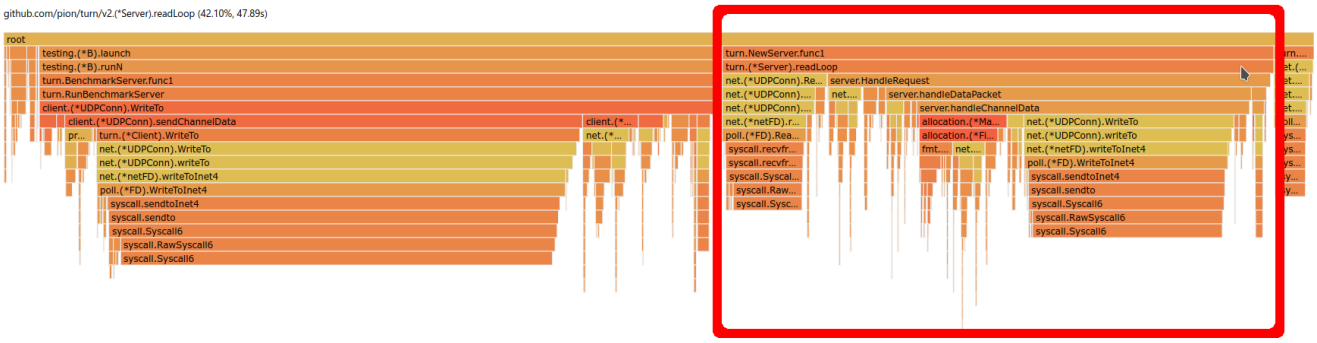

Figure 4: Flame graph of the `pion/turn` benchmark with no offload shows server and client processing; server packet processing logic is highlighted with the red box.

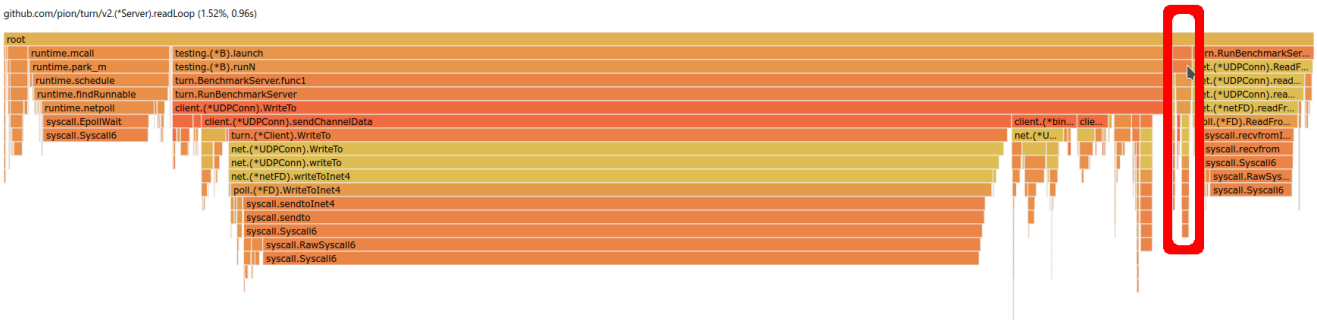

Figure 5: Flame graph of `pion/turn` benchmark with eBPF/XDP offload shows server and client processing; server packet processing logic is highlighted with the red box.

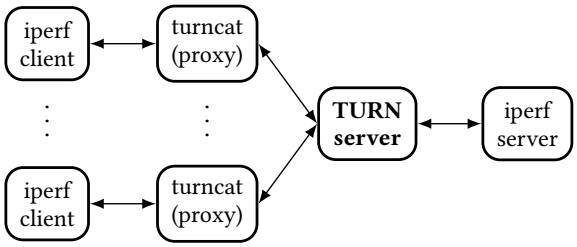

Figure 6: Evaluation Setup: A configurable number of `iperf` clients generate UDP test traffic, one `turncat` TURN proxy per `iperf` client relays the generated UDP traffic to the TURN server under test, and an `iperf` server acts as a peer.

Implementation	Throughput [kpps]	Latency [ms] min	avg	max
1 thread	36.493	3.760	3.944	4.184
4 threads	96.152	0.473	4.311	5.419
XDP	227.378	0.023	0.033	0.074

Table 1: End-to-End results on a single host: 8 clients connects to a peer via the loopback interface.

5.3 End-to-End Performance Evaluation

To get insights into the TURN server performance, we conduct an RFC 2544-style measurement [2]. We deploy the TURN server to a dedicated host, while other components of Fig. 6 run on the other host connected back-to-back to the first one. For each tested TURN server implementation we first identify the maximal throughput without packet loss, as usual with the RFC2544-style measurements, and then we execute 2 consecutive 120-sec measurements using the measured bandwidth. To quantify TURN server overhead, we conducted a *baseline* measurement as well. The baseline eliminates the TURN server and measures pure UDP-forwarding performance with `iptables` at the rate of the measured XDP throughput.

We present results in Table 2. The throughput follows a similar tendency as the microbenchmarks. The XDP offload throughput again reaches the possible maximum (i.e., that of the baseline). The single-threaded TURN server again seems to be producing slightly smaller latency than the 4-thread version. For both cases the minimum delay is at 0.044 ms (not shown in the table). On the contrary, XDP pushes the minimum delay down to 0.028 ms, while on average it is appr. 2–3× smaller than the user-space implementations. Contrasting with the baseline, we found that reason for the slight increase in the minimal and average latency was caused by the first couple of channel messages that got processed by the user-space TURN server *before* the offload became effective. Again,

Implementation	Throughput [kpps]	Latency [ms] avg	Latency [ms] std dev	Jitter [ms]
1 thread	68.5	0.1795	0.2261	0.0265
4 thread	94.8	0.2492	0.3731	0.0351
XDP	134.3	0.0852	0.0994	0.0217
baseline	134.3	0.0386	0.0132	0.0086

Table 2: End-to-End results with 5 clients connecting to a single peer via `pion/turn`, except the *baseline* that implements UDP-forwarding only.

the XDP offload provides a significant improvement to user-space implementations.

To conclude evaluations, we see that adding more socket processing threads to a user-space TURN server improves throughput at the cost of small extra latency. In contrast, the proposed XDP offload not only improves performance significantly, but it also reduces delay meanwhile. This makes our offload a good fit for real-time communication services.

6 RELATED WORK

High-performance media dataplane. A real-time communication service relies an efficient packet processing framework, like Intel DPDK [6], Intel P4/Tofino switches, or Linux eBPF/XDP. Currently, we are not aware of any mature TURN server implementation on top of DPDK. A P4-based media dataplane implementation, which offloads media relaying to P4 switches, is proposed in [7]. Despite the unparalleled efficiency, the requirement for P4-supported hardware limites usability, as opposed to our offload that works on general purpose CPUs and commercial off-the-shelf hardware.

Application-specific eBPF/XDP offload. Outside of network performance improvements, eBPF is proven to be effective for speeding up applications. BMC [5] implements a complex caching solution in eBPF to improve `memcached` performance. Automatically splitting applications to user-space and eBPF components to improve application performance is investigated in [20].

Accelerating network applications with an eBPF/XDP dataplane. XDP and eBPF is used in a wide range of applications to provide high-performance networking capabilities. Among others, load balancers (Katran), Kubernetes gateways (Blixt), firewalls (L4drop, bpftables [11]), container networking interfaces (Cilium, Calico), serverless frameworks (SPRIGHT [15]), software switches (OVS [22]), service mesh components (merbridge, l7mp) use eBPF-based data plane to provide high throughput and low latency following the fastpath/slowpath principle and enabling acceleration for a subset or all of their traffic. Our work aligns with this approach, being the first to accelerate TURN servers.

7 CONCLUSIONS

Real-time communication services are part of our everyday life. TURN services interconnect clients and media servers behind NATs and firewalls. Relaying media streams for large number of users in real-time, TURN servers are notoriously resource and network intensive.

In this paper we introduce an eBPF/XDP offload to improve TURN service performance. We developed a practical XDP offload implementation. Our evaluation shows that the XDP offload improves not just the TURN server performance significantly, but it is also able to improve throughput and reduce delay simultaneously. This makes the offload a good fit for real-time communication services.

Future work focuses on extending the offload engine to support additional eBPF-based mechanisms such as eBPF/tc or sockmap.

ACKNOWLEDGMENT

This work was supported by the ÚNKP-22-4-I-BME-231 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund, and by the NKFIH/OTKA Project #135606. Tamás Lévai is also with the ELKH-BME Information Systems Research Group, and Gábor Rétvári is also with the ELKH-BME Cloud Applications Research Group.

REFERENCES

[1] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. 2021. WebRTC: Real-Time Communication for the Open Web Platform. *Commun. ACM* 64, 8 (jul 2021), 50–54. https://doi.org/10.1145/3453182
[2] Scott Bradner and Jim McQuaid. 1999. Benchmarking Methodology for Network Interconnect Devices. RFC 2544.
[3] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888
[4] David Anderson. 2020. How NAT traversal works. https://tailscale.com/blog/how-nat-traversal-works/
[5] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Berkeley, CA, USA, 487–501. https://www.usenix.org/conference/nsdi21/presentation/ghigoff
[6] Intel. 2023. Data Plane Development Kit. http://dpdk.org.
[7] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. 2020. Offloading Media Traffic to Programmable Data Plane Switches. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. IEEE, New York, NY, USA, 1–7. https://doi.org/10.1109/ICC40277.2020.9149159
[8] l7mp.io. 2023. A Kubernetes media gateway for WebRTC. https://github.com/l7mp/stunner
[9] Tsahi Levent-Levi. 2020. WebRTC TURN: Why you NEED it and when you DON'T need it. https://bloggeek.me/webrtc-turn/
[10] Tamás Lévai et al. 2023. Source Code and Artifacts on GitHub. https://github.com/l7mp/turn/tree/server-ebpf-offload
[11] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a Faster and Scalable Iptables. *SIGCOMM Comput. Commun. Rev.* 49, 3 (nov 2019), 2–17. https://doi.org/10.1145/3371927.3371929
[12] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open VSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 117–130.
[13] Pion. 2023. Pion TURN, an API for building TURN clients and servers. https://github.com/pion/turn
[14] Jon Postel. 1980. User Datagram Protocol. RFC 768.
[15] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259
[16] Tirumaleswar Reddy.K, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. 2020. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 8656.

[17] Anil Rijsinghani. 1994. Computation of the Internet Checksum via Incremental Update. RFC 1624.
[18] C. Partridge R.T. Braden, D.A. Borman. 1988. Computing the Internet Checksum. RFC 1071.
[19] Gábor Rétvári et al. 2023. Implementation of per-client UDP readloops. https://github.com/pion/turn/pull/295
[20] Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. 2023. Automatic Kernel Offload Using BPF. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 143–149. https://doi.org/10.1145/3593856.3595888
[21] Tim Steeves. 2022. WebRTC NAT Traversal Methods: A Case for Embedded TURN. https://www.liveswitch.io/blog/webrtc-nat-traversal-methods-a-case-for-embedded-turn
[22] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open VSwitch Dataplane Ten Years Later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 245–257. https://doi.org/10.1145/3452296.3472914