Taking the Best of Multicast and Unicast with Flexicast QUIC

Louis Navarre
UCLouvain & FRS-FNRS Research Fellow, Belgium
louis.navarre@uclouvain.be

Quentin De Coninck
UMONS, Belgium
quentin.deconinck@umons.ac.be

Tom Barbette
UCLouvain, Belgium
tom.barbette@uclouvain.be

Olivier Bonaventure
UCLouvain & WEL Research Institute, Belgium
olivier.bonaventure@uclouvain.be

ABSTRACT

With IP Multicast, a source can efficiently send the same information to a set of receivers attached to a multicast tree. Unfortunately, when distributing live video or large files, some receivers might be unable to join the multicast tree. Applications willing to use multicast for efficiency must also support unicast to reach all their receivers. Given the complexity of mixing unicast and multicast, most popular applications only use unicast protocols.

The large deployment of QUIC, a secure and flexible transport protocol that runs above UDP, allows for reconsidering multicast at the transport layer. We design and implement *Flexicast QUIC*, an extension of Multipath QUIC that enables applications to use multicast where and when it works efficiently and seamlessly fall back on unicast otherwise. Our in-lab performance evaluation shows that a Flexicast QUIC source can sustain up to 1000 receivers for an aggregated traffic of more than 80 Gbps, more than 4 times what we achieve with (unicast) QUIC in the same setup. We also show that Flexicast QUIC can easily distribute a video stream and recover from transient failures on the underlying multicast tree while maintaining excellent quality of experience.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Network protocol design**; **Network performance analysis**;

KEYWORDS

QUIC, Multipath, Multicast, Flexicast

1 INTRODUCTION

In recent years, one-to-many applications such as low latency video-streaming and software deliveries have become omnipresent, and the demand for such applications is expected to continue growing in the coming years [41]. Applications use unicast protocols to distribute this content to multiple receivers, creating per-receiver flows and imposing strong pressure on the network and the sender. For example, Akamai, one of the world's largest content providers, reached peak traffic in 2022 of 250 Tbps of data [4]. A more efficient substitute to unicast is to distribute this content using multicast protocols. With multicast, the source generates a single copy of each packet, which routers replicate throughout the network to reach all receivers. Akamai showed in 2020 that between 20 % and 50 % of its traffic could benefit from multicast [32].

Deering first proposed multicast in IP-based networks [20, 22]. This proposal attracted a lot of interest from network researchers who designed multicast applications [45], multicast routing protocols [21] and deployed them on the MBONE [5, 26]. The experiments conducted over the MBONE, including the live distribution of events such as SIGCOMM conferences and IETF meetings, influenced many techniques used today by audio and video applications. Today, the main wide-area multicast applications are IP TV in ISP networks [43, 46], some file distribution services in enterprises [47] and specialized financial applications [14]. However, multicast is still not mainstream. One of the reasons IP Multicast is not globally available today is that it is difficult for ISPs to monetize multicast across inter-domain boundaries [23]. Another reason is that none of the standardized Internet transport protocols supports multicast.

Recent advancements in network-layer multicast mitigate the deployment problems of multicast routing, such as the Bit Index Explicit Replication [56] and Automatic Multicast Tunneling [10]. However, network support for multicast is still not widely deployed, and applications wishing to benefit from multicast must implement an alternative delivery if specific receivers do not support it.

This paper introduces the concept of flexible multicast, i.e., *Flexicast*. A Flexicast transport protocol benefits from efficient multicast distribution when possible and leverages unicast fall-back when multicast is not available or under-performing. We leverage QUIC [37], which combines the security features of TLS [50] with the reliability and congestion control features of modern TCP [25], in addition to stream multiplexing and connection migration features. The QUIC protocol is already largely deployed on the Internet [57, 58], and about twenty known implementations co-exist.

We build upon QUIC and its Multipath extension [42] to design *Flexicast QUIC*. The high-level architecture of Flexicast QUIC is illustrated in Figure 1. Multipath QUIC enables end-hosts to create multiple bidirectional paths within the same connection. Flexicast QUIC extends this design by suggesting that additional paths may be *unidirectional*, similarly to [19]. In addition, we use different sets of cryptographic keys for each additional path. A Flexicast QUIC connection thus exposes per-receiver bidirectional unicast paths and shared unidirectional *flexicast flows* within the same protocol. While the unicast paths ensure a fall-back and secure channel, the flexicast flows can be distributed in a multicast network for efficient and scalable data delivery to a group of receivers. Applications can leverage Flexicast QUIC since a *flexicast flow* is viewed as *another path* with different cryptographic keys.

A Flexicast QUIC source primarily sends data on the *flexicast flows* to improve efficiency and scalability. If one of the receivers cannot receive the data on the flow, e.g., due to congestion or because it is in a part of the network where multicast has not been

Accepted for publication on March 14, 2025

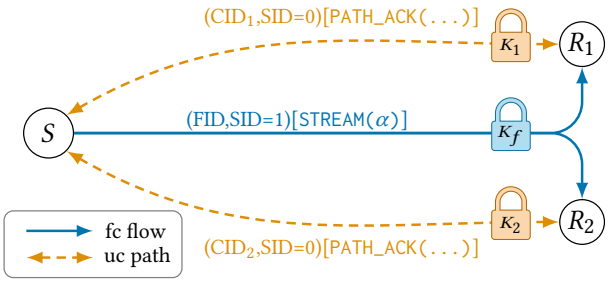

Figure 1: High-level overview of Flexicast QUIC. A connection combines per-receiver (R_1 and R_2) bidirectional unicast paths and shared unidirectional *flexicast flow(s)*. The unicast paths are protected with individual keys ($K_{\{1,2\}}$) while the flexicast flow uses a common key, K_f. Flexicast QUIC leverages Multipath QUIC [42] to manage the flexicast flow as an additional path. While a QUIC path is identified by a Connection ID (CID), flexicast flows use a Flow ID (FID). Each path/flow uses its own packet number space ID (SID).

(correctly) configured, the source can decide to send or retransmit data over this receiver-specific unicast path.

Contributions. We make the following contributions:

- We design Flexicast QUIC (FCQUIC), relying on and extending Multipath QUIC [42].
- We implement of FCQUIC in Cloudflare *quiche* [15].
- Our benchmarks show that an FCQUIC source can sustain more than 1000 receivers and deliver >80 Gbps of traffic.
- Our evaluations show the robustness of Flexicast QUIC when the multicast network fails.

We release the source code of Flexicast QUIC[1] to explore larger-scale deployment as part of our future work (Section 8).

2 BACKGROUND

This section gives a brief overview of QUIC [37], Multipath QUIC (MPQUIC) [42] and IP Multicast [30].

QUIC. QUIC [37] is a connection-oriented protocol running above UDP. TLS 1.3 is embedded in QUIC to provide a fast and secure session handshake. A QUIC connection starts with a handshake, during which the TLS session keys are computed and connection parameters negotiated. In contrast with the 4-tuple used by TCP, a QUIC connection is identified by source and destination Connection IDs (CID) chosen by the end-hosts. The destination CID is part of the header of each packet. Besides a few flags, this is the only information not encrypted in QUIC packets. Each QUIC packet is identified with a monotonically increasing packet number (PN). This packet number is also encrypted in each QUIC packet. QUIC packets are *frame containers*, i.e., they carry control and data frames. This architecture makes it easy to extend the protocol by defining new frames. QUIC supports reliable, ordered stream multiplexing through the `STREAM` frame and unreliable communication with the `DATAGRAM` frame. Retransmitted frames are sent in new QUIC packets with increased packet numbers.

[1]https://github.com/IPNetworkingLab/flexicast-quic

Multipath QUIC. Multipath QUIC [42] extends QUIC by allowing to simultaneously use multiple paths within a single QUIC connection. Multipath QUIC associates each path with a distinct set of CIDs, communicated using the `PATH_NEW_CONNECTION_ID` frame. Multipath QUIC support is negotiated during the handshake with the `initial_max_path_id` transport parameter. Each path on a Multipath QUIC connection uses a different packet number space [18]. Concretely, this means that packets sent on each path are totally decoupled from the others. Some data sent over one path can be retransmitted over another path. Furthermore, a receiver can acknowledge packets received on one path over another path. The IETF is finalizing the standardization of this extension [42]. Several open-source implementations of QUIC already support multipath[2].

IP Multicast. Enterprise and ISP networks supporting multicast usually rely on Source-Specific Multicast (SSM) [30]. SSM allows a single source to send data over multicast trees. The tree is identified by the address of the source and a multicast destination address. The receivers use MLD [17] or IGMP [11] to join the multicast tree dynamically. Multicast-enabled applications use either proprietary protocols or RTP [51].

3 FLEXIBLE MULTICAST

This paper proposes the concept of *flexible multicast* (*Flexicast* in short). We refer to Flexicast as the ability for a source to efficiently send encrypted and authenticated data to a set of N receivers. For this, the source manages $N + 1$ cryptographic keys. First, it negotiates key K_i to securely exchange packets with receiver R_i, creating a bidirectional, secure *unicast path*. Second, it creates a shared key, K_f, that it communicates to all receivers using this secure unicast path. The source uses this key to encrypt packets targeting all receivers simultaneously, thus creating a shared unidirectional flow of data that we call a *flexicast flow* thanks to the way packets sent on this flow can be delivered to the receivers: either through an underlying multicast tree or using replication with IP unicast.

Flexicast flow. If all receivers are attached to a multicast distribution tree, the source can encrypt and authenticate a single packet using K_f, and rely on the network to efficiently distribute it to all receivers. Since all receivers have received K_f from the source, they can decrypt packets sent on the flexicast flow. However, there are situations where only a fraction of the receivers have managed to join the multicast tree. Consider that receiver R_1 could not join the multicast tree. In addition to the packets sent on the flexicast flow, the source will generate and encrypt a new packet for R_1 using K_1 with the same application data.

If $m \leq N$ receivers did not manage to join the multicast tree, the source needs to encrypt and authenticate $m + 1$ packets for each data chunk it sends. When $m = N$, this method would be equivalent to delivering data over (unicast) QUIC, which is the current solution QUIC servers use nowadays [32]. However, measurement studies [38, 53] showed that the generation and encryption of QUIC packets take substantial CPU resources on servers.

Flexicast offers a more efficient and elegant solution. Considering the worst case, i.e., $m = N$, the source could still generate a single packet and encrypt it using K_f. Instead of relying on an underlying

[2]https://github.com/quicwg/multipath/wiki/QUIC-Implementations-with-multipath-support

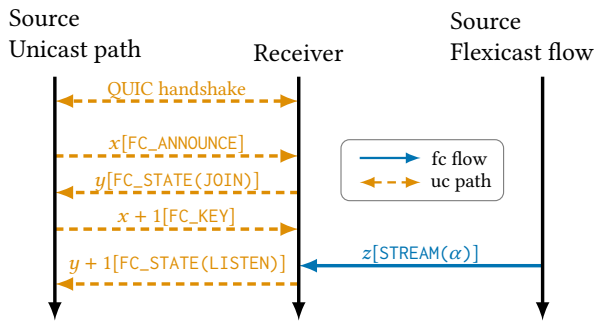

Figure 2: After the QUIC handshake, the source advertises its available flexicast flow. If the receiver joins it, the source sends the decryption key of the flow. Once the receiver starts receiving packets from the flexicast flow, it notifies the source through the `FC_STATE(LISTEN)` frame. Packet are represented as $\{pn\}$[`FRAMES`] with pn being the packet number.

multicast distribution tree, the source could duplicate the bytes generated by the transport protocol and send them over IP unicast to each receiver. The `sendmmsg` system call can efficiently perform such action.

Flexible multicast. The source maintains per-receiver unicast paths during the lifetime of the connection. This path offers a secure fall-back mechanism whenever (i) the underlying multicast distribution tree fails or (ii) the receiver becomes a bottleneck, e.g., it cannot follow the pace of the flexicast flow. Such a receiver can leave the flexicast flow and seamlessly fall back on unicast within the same connection, thus offering *flexible* multicast to the endpoints: multicast for its efficiency and unicast for its robustness.

4 FLEXICAST QUIC

This section presents the design of Flexicast QUIC (FCQUIC), an extension of Multipath QUIC [42] to support flexicast. FCQUIC relies on Multipath QUIC (MPQUIC) for the creation, management and scheduling of the flexicast flows.

A Flexicast QUIC connection starts as a regular Multipath QUIC connection besides the utilization of a flexicast transport parameter. During the handshake, the source and the receiver establish a first bidirectional path that is protected by one set of TLS keys. Subsequent receivers connecting to the source will also establish their own bidirectional paths protected by other sets of TLS keys.

We describe the mechanisms used by FCQUIC: (i) how the source advertises a flexicast flow; (ii) how the source uses and advertises the shared key; (iii) the extensions to MPQUIC; (iv) how FCQUIC leverages the reliability mechanisms of QUIC; and (v) how FCQUIC endpoints manage their membership to a flexicast flow.

4.1 Advertising a Flexicast Flow

Figure 2 illustrates the first steps of a Flexicast QUIC connection, which starts with a standard QUIC handshake [37], creating the *bidirectional unicast path*. During this handshake, both endpoints advertise their local support of flexicast with the `enable_flexicast` transport parameter and the `initial_max_path_id` transport parameter required for Multipath QUIC.

Advertisement. After the QUIC handshake, the source announces the available flexicast flow using the `FC_ANNOUNCE` frame. A flexicast flow is identified by a destination Connection ID (CID) chosen by the source. For clarity, we call the destination CID of the flexicast flow the *Flow ID* (FID). The `FC_ANNOUNCE` frame contains the FID, the source and multicast IP addresses, and the UDP destination port number of packets sent on the flexicast flow[3].

Joining a flexicast flow. A receiver sends to the source an `FC_STATE` frame with the `JOIN` action to request to join the advertised flexicast flow. At this point, the receiver creates a state for the new flexicast flow. The source answers by sending the `FC_KEY` frame containing the shared key it uses to encrypt the packets sent on the flexicast flow. The receiver then joins the underlying multicast tree, e.g., using IGMP [11] or MLD [17]. Once the receiver starts receiving packets on the flexicast flow, it means that the underlying multicast tree is working, and the receiver sends an `FC_STATE` frame with the `LISTENING` action. If multicast is not available, the receiver falls back on unicast. The `FC*` frames are exchanged over the unicast path and are thus secured using the (unicast) TLS keys negotiated during connection establishment.

4.2 Changes to Multipath QUIC

The current version of MPQUIC [42] uses a single cryptographic context for the entire connection, i.e., all paths use the same key[4]. FCQUIC extends MPQUIC by providing a cryptographic context for each path. The flexicast flow uses a key chosen by the source and communicated to all receivers, while each unicast path is protected using a different key for each receiver. Flexicast QUIC also requires the flexicast flow to be *unidirectional* once established. Whilst this is compliant with MPQUIC's design [42], FCQUIC has specific considerations: a receiver sends *all* its packets on the unicast path and only uses the flexicast flow to receive packets. The server never receives packets from the flexicast flow. It only sends on the flexicast flow frames intended for all receivers.

Thanks to Multipath QUIC, receivers view the flexicast flow as a second MPQUIC path with a different decryption key. Because MPQUIC [42] uses a *different* packet number space per path, there is no coupling between the packets sent on both paths. The unicast path is mainly used for *control* frames, while the flexicast flow carries the *data* frames. The source can also retransmit frames over the unicast path to recover from losses. Receivers with a failing flexicast flow can fall back on unicast to receive subsequent data.

The third change relates to the creation of a flexicast flow. In Multipath QUIC, a host must first advertise new Connection IDs. Then, it initiates a new path by exchanging `PATH_CHALLENGE` and `PATH_RESPONSE` frames. This prevents attacks by verifying that the remote endpoint can reply to packets sent on the new address [42]. In MPQUIC, each new path is bound to a set of CIDs. In FCQUIC, since the destination address of the flexicast flow is a multicast IP address, the receiver cannot send `PATH_CHALLENGE` frames from this address. Thus, the receiver creates a state for the new path without actually sending path-related frames. Moreover, the receiver's source CID for this new path is the Flow ID advertised in the `FC_ANNOUNCE` frame. In Flexicast QUIC, since the source forwards this information on the secure unicast path, the receiver

[3]We assume that the receiver does not already use either the FID as a source CID or the UDP port.
[4]Packets are encrypted and authenticated using AEAD with a Nonce whose value depends on the path identifier, but the encryption key remains the same.

trusts the advertised multicast address. Additionally, because the flexicast flow is unidirectional, an external attacker cannot spoof the address of the receiver to send malicious data to the source.

4.3 Reliability mechanisms

By default, a QUIC receiver should send an ACK frame after receiving at least two ack-eliciting packets [37]. The default maximum delay between two acknowledgments is set to 25 ms [37].

Multipath QUIC extends this procedure by replacing ACK frames with PATH_ACK frames [42]. These frames include the *space identifier*, allowing a receiver to acknowledge on path X data received on path Y. MPQUIC allows a sender to retransmit data over a different path than the initially used one. FCQUIC leverages multipath to ensure full reliability by retransmitting frames either on the flexicast flow or the unicast path, depending on how many receivers have reported a loss.

To acknowledge data received on the flexicast flow, FCQUIC receivers send PATH_ACK frames on their unicast path with the *space identifier* of the flexicast flow. To prevent ACK implosion, an FCQUIC source advertises a *minimum ack delay* inside the FC_ANNOUNCE frame. We analyze the impact of this parameter on the scalability of FCQUIC in Section 6.1.

The source releases memory for a packet sent on the flexicast flow once all receivers have acknowledged it, or it is considered lost [35]. If the packet is lost, the flexicast flow retransmits its frames on the unicast paths of the receivers who did not acknowledge the corresponding packet. As such, the computed RTT on the flexicast flow is a function of *slowest* receiver to acknowledge a given packet number.

Congestion control. Because the flexicast flow is shared, its throughput depends on all receivers. The FCQUIC source maintains a per-receiver congestion state for the data sent on the flexicast flow. These congestion states are adapted using the per-receiver acknowledgments, leveraging existing unicast congestion control algorithms to determine the appropriate sending rate. The current version of FCQUIC sets the throughput on the flexicast flow as the minimum rate among all active receivers from the flow. As such, the source adapts its transmission rate to the slowest receiver. Thanks to Multipath QUIC, the congestion state of the flexicast flow is independent of the unicast path state because the source only uses the PATH_ACK frames acknowledging packets sent on the flow. The source can remove receivers with insufficient performance (e.g., those with a too-low reception bit rate) from the flexicast flow and continue sending data through their unicast path. Managing the flexicast flow ensures a minimum quality of experience and lets the source deal with potentially malicious receivers.

4.4 Managing the Flexicast flow

Both the Flexicast QUIC source and receivers manage the status of the flexicast flow using the FC_STATE frame. Receivers use the LEAVE action to leave the flexicast flow. Upon reception, the source falls back on unicast for this receiver, using the unicast path and standard QUIC for data delivery. The source can also *unilaterally* eject a receiver from the flexicast flow by sending an FC_STATE frame with the LEAVE action. Once the source sends this frame, it does not consider the receiver in the flexicast flow and falls back on unicast. For example, this event can occur if the receiver reported too many losses compared to the other receivers, meaning that the flexicast flow would underperform by keeping this receiver.

There are also situations where a working flexicast flow might stop working for some receivers. For example, a link failure might reroute some branches of the multicast tree over routers that do not support multicast, resulting in a black hole for some receivers. Section 6.2 explores this scenario and shows how a Flexicast QUIC source deals with such failures.

5 IMPLEMENTATION

We implement Flexicast QUIC inside the multipath branch [16] of Cloudflare *quiche* [15] written in the Rust programming language. Our implementation adds ~10,000 lines of code (LoC) and 5000 LoC of tests. Our multithread applications based on `tokio` [52] required ~4500 LoC. The FCQUIC receiver application is relatively similar to a Multipath QUIC equivalent since the flexicast flow is considered as a unidirectional *new* path. When it receives a packet with the Flexicast Flow ID as destination CID, it uses the shared decryption key (K_f). The implementation modifies the packet scheduler to forbid receivers from sending packets using the flexicast flow.

The main challenge is to design a scalable architecture on the Flexicast QUIC source with a shared flexicast flow where information from each receiver impacts the state of the flexicast flow while still ensuring scalability to large groups of receivers and avoiding the ACK implosion problem. This Section describes how we implemented the communication between the per-receiver unicast paths and the shared flexicast flow on the source to ensure scalability. Figure 3 presents an overview of this architecture considering four receivers listening to the same content. This architecture is composed of 4 elements.

I/O. The *I/O* module receives and sends packets. When receiving a new packet, the module verifies the destination CID. If known, the packet is forwarded to the associated unicast path connection; otherwise, this is a new connection handshake that the source will process. Upon handshake completion, a new unicast connection instance is created.

Unicast paths. The unicast connection instances (*UC*) handle the *unicast paths*. In Figure 3, they correspond to the packet number space ID (SID) 0. This module reads acknowledgments from the receivers (from the PATH_ACK frames) and forwards them to the controller. It also handles the unicast retransmission of frames lost on the flexicast flow. The *UC* module checks the liveliness of the flexicast flow and, when relevant, notifies the controller that the corresponding receiver should fall back on unicast; in that case, it will continue sending new data to the receiver on the unicast path. The unicast connections use per-receiver connection keys (e.g., K_1) for packets from the unicast path.

Flexicast flow. The flexicast flow (*FC Flow*) receives data from the application, either using streams or datagrams, and generates packets on the flexicast flow, using the shared key (K_f) and a different SID. It also forwards application data to the controller if some receivers fall back. The module uses the aggregated acknowledgments from the controller to delegate unicast retransmission.

Controller. Finally, the *Controller* module binds the unicast paths and the flexicast flow. Its role is to aggregate information (e.g., acknowledgments) from the receivers and dispatch the unicast retransmissions to the receivers. This module provides scalability

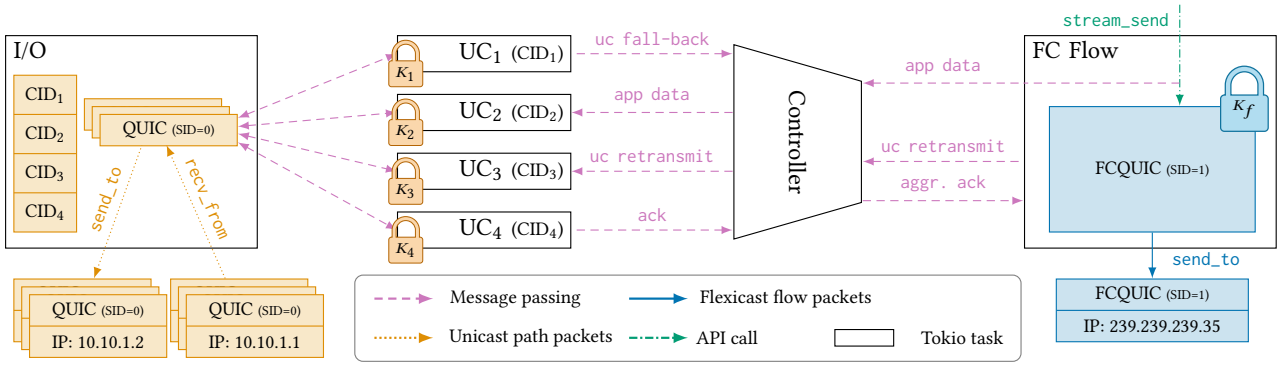

Figure 3: Overview of our implementation of the Flexicast QUIC source.

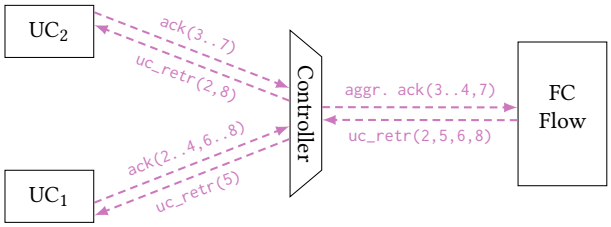

Figure 4: The controller aggregates acknowledgments received on the unicast path before sending them to the flexicast flow module. It delegates unicast retransmissions to the controller, which dispatches them to the unicast path based on the per-receiver acknowledgments.

by handling per-receiver information and only communicating with the *FC flow* module with aggregated updates. If a receiver falls back on unicast, the *Controller* is also responsible for buffering application data if the unicast path instance cannot sustain it.

Multithread architecture. Our implementation leverages the `tokio` [52] runtime to execute the different modules on multiple cores on the same machine, using message passing to communicate. Future work might consider running the modules on distinct machines as fan-out servers [28] and design specific protocols to communicate using the same control messages as this paper.

Scalable reliability. All active receivers must acknowledge each packet sent on the flexicast flow before the source can release it from memory. This raises a challenge regarding how to manage the acknowledgments while ensuring scalability. The *Controller* aggregates acknowledgments from each receiver and sends a single acknowledgment for each packet to the flexicast flow module. It then distributes the unicast retransmissions using these acknowledgments. Figure 4 shows an example of acknowledgment aggregation and unicast retransmission with two receivers. In this example, the first UC_1 instance receives from the receiver acknowledgments for packet numbers 2 to 4 and 6 to 8; the second receives from 3 to 7. They forward these acknowledgments to the *Controller*, which aggregates them before sending them to the *FC Flow* module. The *FC Flow* module delegates the frames sent in packets that are considered lost to at least a receiver (2,5,6,8). The *Controller* dispatches the retransmissions to each unicast path based on the per-receiver acknowledgments.

6 EVALUATION

We evaluate our implementation based on *quiche* on both CloudLab [24] and emulated networks, respectively, to benchmark its scalability and to assess its robustness in case of failures on the underlying multicast distribution tree. All our experiments use the Network Performance Framework [7] and are reproducible on CloudLab and commodity servers.

6.1 Scalability of FCQUIC

CloudLab topology. We leverage CloudLab [24] to benchmark Flexicast QUIC. We use 6 *d6515* nodes: the Source, the Network, and four nodes connected on a LAN with the Network, emulating receivers. All links have a capacity of 100 Gbps, and the nodes are equipped with AMD EPYC 7452 processors, 32 CPU cores, 64 threads, and 128 GB of DRAM. We run up to 250 receivers per host in distinct network namespaces for 1000 receivers.

FastClick router. We emulate the multicast tree on a single node (the Network) using FastClick [8], an extension of the Click modular router [40] enabling fast packet processing through optimizations such as packet batching and DPDK [27] support. The FastClick router replicates packets to each receiver using multiple cores to simulate the multicast distribution tree. This implementation sustains 95 Gbps of replicated traffic. It also forwards unicast packets between the Source and the receivers.

Measured metrics. The Source generates a constant 80 Mbps bit-rate, which is considered an upper-bound for a 4K video streaming application [3]. The payload consists of packets with 1200 B of payload. We measure three metrics while increasing the number of simultaneous receivers:

- *Recv throughput*: the throughput (i.e., including protocol overhead), aggregated on all receivers;
- *Max source CPU*: each second, we poll the CPU usages on the Source and report the busiest CPU to identify a potential bottleneck in our implementation;
- *Ack rate*: the throughput of the acknowledgments returned by all receivers.

We compare these metrics with (i) *Baseline*: pure UDP traffic without acknowledgments; (ii) *QUIC*: per-receiver unicast QUIC connections. We load-balance the receivers between 4 parallel I/O loops to improve the scalability of QUIC; (iii) *FC*: Flexicast QUIC; (iv) *FC-5*:

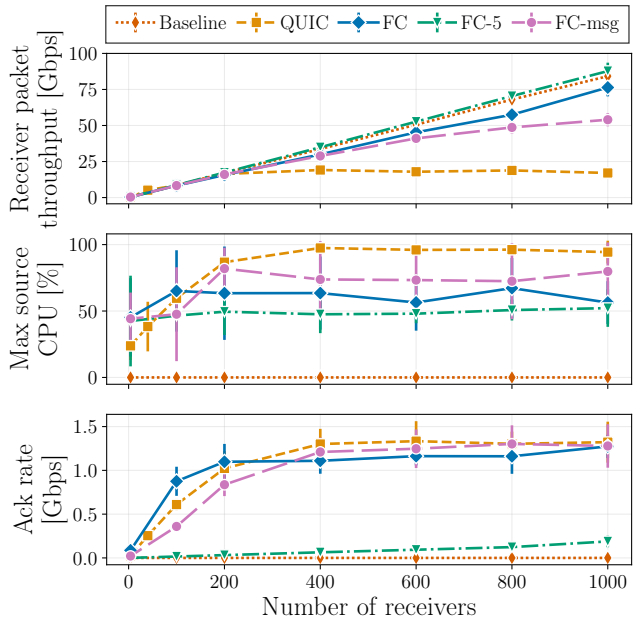

Figure 5: Benchmark results using CloudLab [24]. The source sends 1200 B streams at 80 Mbps. The Network is emulated using our Fastclick [8] router. We run a UDP Baseline, QUIC, and FCQUIC.

Flexicast QUIC with an *ack delay* of 5 ms; and (v) *FC-msg*: Flexicast QUIC using the `sendmmsg` system call to replicate packets at the Source instead of relying on an underlying multicast distribution tree. For unicast QUIC, we run four server instances in parallel and evenly split the receivers among these instances to improve its scalability. We disable the flow and congestion controllers (both for QUIC and FCQUIC) to push the implementation to its limits.

Unicast QUIC limits. Figure 5 presents the three metrics in the five aforementioned scenarios. The *Baseline* gives an ideal target cumulated throughput of ~80 Gbps of downwards traffic for 1000 receivers. With (unicast) QUIC, the Source struggles to deliver the content to more than 200 receivers for an aggregated throughput of ~20 Gbps. The source CPU usage reaches full utilization, explaining the cap at ~20 Gbps. **More precisely, all 64 available threads reached 100 % utilization after 200 receivers.** As reported in other studies [39, 53], the cost of I/O, packet encryption, and connection management is significant, thus becoming the bottleneck when considering hundreds of receivers.

FCQUIC supports 1000 receivers and delivers more than 78 Gbps of traffic. Using Flexicast QUIC (the `FC` curve), the Source must only generate, encrypt, and send a single copy of each packet on the flexicast flow. As such, this process is independent of the number of receivers, and the Source can send data to numerous recipients as long as the Network can sustain it. The maximum CPU usage of the FCQUIC source remains acceptable (between 50 % and 75 %). We also noted that the median CPU usage remained under 20 % even for 1000 receivers.

Acknowledgment rate. A famous problem regarding reliable multicast is the ack implosion problem [49]. The ack implosion problem occurs when the Source becomes the bottleneck when dealing with all the acknowledgments returned by the receivers. Because our *Controller* aggregates acknowledgments before they are sent to the *FC Flow*, sending new packets on the flexicast flow scales to numerous receivers.

Figure 5 reports the *ack rate* of the receivers towards the Source on the unicast path. Theoretically, this *ack rate* should linearly increase with the number of receivers. We see that after 200 receivers, both QUIC and FCQUIC *ack rates* saturate at ~1.4 Gbps. Multiple factors related to the experimental setup explain this limit.

Limits of the experimental setup. We run all receivers on four distinct 64-thread nodes. For ≥ 400 receivers, we simultaneously run ≥ 100 receivers on the same machine. As a consequence, we notice contention between receivers sharing the same CPU. We noticed that receivers started processing packets per batch when resources became limited, hence decreasing the *ack rate* because a single `PATH_ACK` was sent for multiple received packets.

We identified a second limit in the FastClick router emulating the Network. We had to add a token bucket using `tc` to pace the source at 100 Mbps to avoid overloading the FastClick router. Since the Flexicast QUIC Source also has to send control information for each receiver (e.g., to acknowledge packets from the receivers), these packets compete with the data packets on the flexicast flow. This limit explains the gap between the `Baseline` and `FC` curve for ≥ 600 receivers at the top of Figure 5. However, even without these limits, we expect FCQUIC to scale to 1000 receivers thanks to the aggregation of the *Controller*.

Adding an ack delay. We run the same benchmark with an *ack delay* [36] of 5 ms. The Source advertises this ack delay in the `FC_ANNOUNCE` frame. For 1000 receivers, the cumulated *ack rate* remains below 200 Mbps, far below the value without any ack delay. Because the ack delay reduces the packets that the receivers generate, the Source must also generate fewer packets on the flexicast flow. As such, the cumulated throughput follows the baseline expectations and allows FCQUIC to reach the ~80 Gbps of our UDP Baseline. We even notice that the throughput with `FC-5` is higher than the *Baseline*, which is expected due to the byte overhead of QUIC.

Using *sendmmsg* in non-multicast networks. As explained in Section 3, one could still benefit from FCQUIC even in a non-multicast capable network. Instead of generating, encrypting, and sending N QUIC packets to N receivers, an FCQUIC source can generate and encrypt a single packet on the flexicast flow. To distribute these packets, the Source can *replicate* and forward them as unicast packets. Doing so reduces CPU usage of the Source since replicating packets is much cheaper than generating new ones. We use the `sendmmsg` system call to replicate the packets inside the kernel with low overhead. These `sendmmsg` calls are performed by dedicated threads. The `FC-msg` curve on Figure 5 shows that by using `sendmmsg` our FCQUIC Source supports much more receivers than QUIC, with a much-reduced CPU load. The maximum CPU usage oscillates between 50 % and 90 %, while the median CPU usage remains below 50 % even for 1000 receivers. We cannot reach the same throughput as FCQUIC inside a multicast-capable network since replicating packets at the Source still costs more than using a real multicast tree.

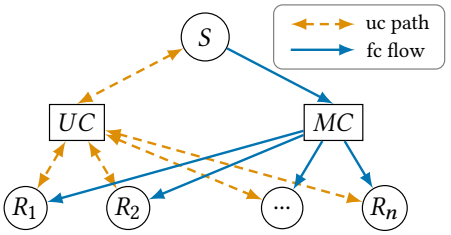

Figure 6: Topology used to assess the robustness of Flexicast QUIC. We emulate 20 receivers. *UC* and *MC* are two routers connecting in a full mesh to the receivers.

6.2 Robustness of Flexicast QUIC

Emulated topology. We now evaluate the robustness of Flexicast QUIC when sending a video stream in an unstable multicast network. We emulate the network topology illustrated in Figure 6 on an Ampere(R) Altra(R) Processor Q80-30 CPU at 2.8 GHz. Each node lies in its dedicated network namespace, and we use `tc` to set a bandwidth limit of 10 Gbps and 10 ms latency on each link connected to the 20 receivers, i.e., the RTT between each receiver and the Source is ~20 ms. Endpoints use the CUBIC congestion controller on the unicast path and the flexicast flow.

5 Mbps video stream. We use FFmpeg [1] and GStreamer [2] to generate the video stream and display the video to the user. The video stream is a 180 s replay of drone remote piloting videos generated by Baltaci *et al.* [6]. The video is encoded using the H.264 codec [55] with a constant rate of 5 Mbps and low-latency parameters. The Source uses FFmpeg to send the video as an RTP [51] stream. The receivers run GStreamer to consume the video stream. We compute the Structured Similarity (SSIM) [54]. This state-of-the-art metric measures the similarity between each video frame received by the receiver and the frame generated by the Source. A value of 1 indicates that both frames are exactly the same. We also report the *frame latency*, i.e, the time it takes for each frame from the GStreamer source to the FFmpeg sinks.

Failure scenarios. From an operational viewpoint, IP Multicast is more fragile than IP unicast. To enable IP Multicast in a network, the operator needs to enable it on each network interface on each router. If they forget one of these commands, multicast tree establishment will fail when the shortest path to the Source passes through this link. Furthermore, some routers have limitations on the amount of multicast state they can store. Such routers can discard a request to join a multicast tree from a downstream router. In a multicast network, unicast may work perfectly, while some branches of a multicast tree may not. To model this scenario, we disable links of the multicast tree. Every 5 seconds, we randomly select one of the 20 links between the *MC* router and the receivers and disable it for 15 seconds with a probability of 50 %. Multiple links may be disabled at the same time. We use a seed to reproduce the failure pattern across experiments.

Aliveness of the flexicast flow. We implement a packet scheduler on the Source to detect the failure of the flexicast flow for each receiver. Whenever there are bytes in flight on the flexicast flow, the scheduler verifies that it gets *new* acknowledgments from the receiver for data sent on the flow within the *fall-back delay* (D_{fb}). If the receiver does not send *new* acknowledgments within D_{fb}, or if the acknowledgments concern older data, the scheduler considers that the flexicast flow is failing, and the Source falls back

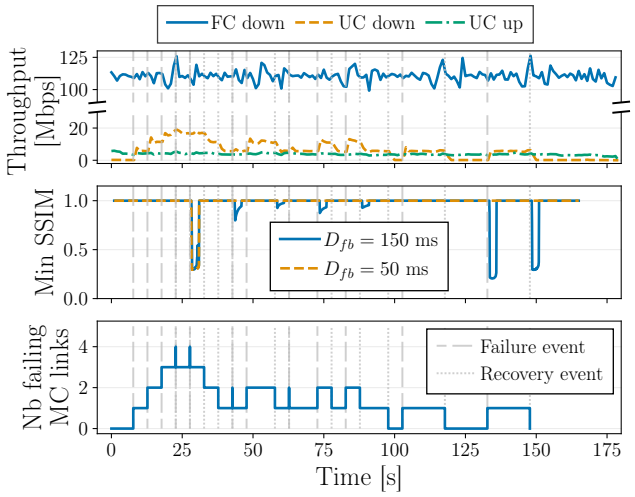

Figure 7: Evolution of the throughput, *minimum* SSIM among all receivers for each frame, and the number of failing multicast links for G_{pb} = 100 ms and D_{fb} = 150 ms. Each vertical bar represents a failure or recovery event.

on unicast delivery for this receiver. Unacknowledged data is retransmitted on the unicast path, and subsequent data packets are sent on that path. Once the receiver starts sending again `PATH_ACK` frames concerning (new) data received on the flexicast flow, the scheduler considers it back alive, and the receiver comes back in the flexicast flow. The objective of our packet scheduler is to provide *seamless* transition between unicast and multicast whenever some link is failing. We evaluate values of the fall-back delay (D_{fb}) and the GStreamer playback buffer (G_{pb}) in {50, 100, 150} ms.

Figure 7 shows the source rates with G_{pb} = 100 ms and D_{fb} = {50, 150} ms. We measure the cumulated bit-rate (i) on the flexicast flow (`FC down`); (ii) from the source to the receivers (`UC down`) and (iii) from the receivers to the source (`UC up`) on the unicast paths. The vertical lines mark events of failure/recovery of the multicast links. The middle graph shows the *minimum* SSIM among all receivers for each video frame. The bottom graph shows the number of failing links at any moment during the experiment.

The Flexicast QUIC Source correctly falls back when it detects that the multicast tree is failing for some receivers without impacting other receivers. Since impacted receivers cannot benefit from efficient multicast delivery, the Source must transmit data using unicast. When the Source detects the failure of such a receiver, it no longer considers this receiver in the flexicast flow. As such, this receiver cannot be selected as the slowest member when computing the congestion window of the flexicast flow. This results in a steady bit rate on the flexicast flow for other receivers despite the failure of this receiver. We notice no significant difference in the throughput with D_{fb} = 50 ms.

The fall-back mechanism keeps video stream quality even if the multicast tree is failing. Figure 8 shows the Structured Similarity (SSIM) for each video frame on each receiver in the nine combinations of D_{fb} and G_{pb}. Results show that the SSIM remains perfect in ~99.4 % of the time, showing that multicast-failing receivers do not significantly impact the others. As expected,

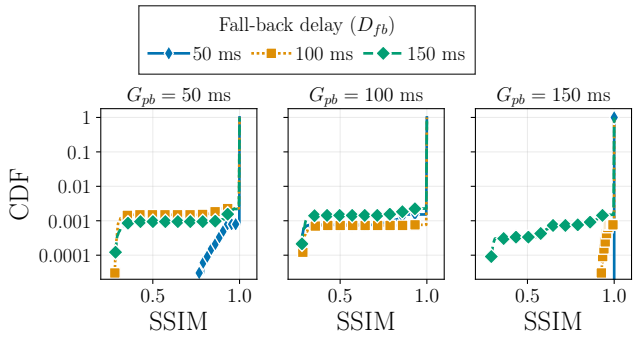

Figure 8: Structured similarity (SSIM) aggregated on all receivers for $G_{pb} = \{50, 100, 150\}$ ms and $D_{fb} = \{50, 100, 150\}$ ms.

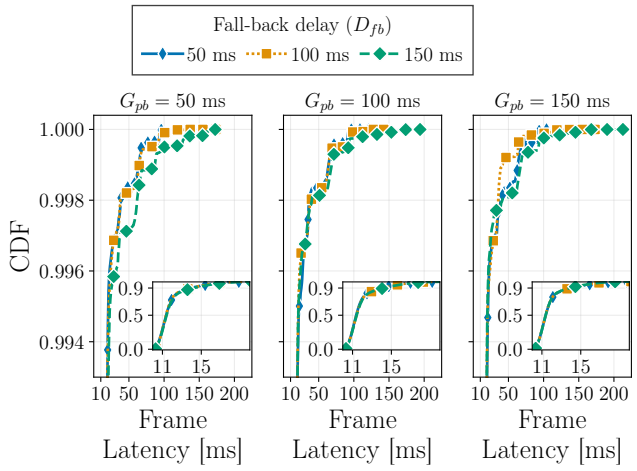

Figure 9: Frame latency aggregated on all receivers for $G_{pb} = \{50, 100, 150\}$ ms and $D_{fb} = \{50, 100, 150\}$ ms.

the difference between the fall-back delay (D_{fb}) and the GStreamer playback buffer (G_{pb}) has an impact on the SSIM. Whenever $D_{fb} < G_{pb}$, the Flexicast QUIC scheduler can quickly detect a multicast failure, fall-back, and retransmit lost frames on the unicast path *before* the expiration of the GStreamer playback buffer, thus keeping a higher video quality. If $D_{fb} \geq G_{pb}$, the failure may be detected too late to retransmit lost frames on the unicast path.

Figure 9 reports the frame latency for the same experiments, aggregated on all receivers. The inner Figure shows the main cumulative distribution, while the outer highlights the tail latency. While the median latency remains close to the 10 ms one-way delay, we notice that it increases up to 50 ms for 0.2 % of the frames. This increase results from the time required to detect the link failure and retransmit in-flight frames on the unicast path. However, only a small fraction of frames (>0.01 %) have a latency above 150 ms when D_{fb} =150 ms. For smaller values of D_{fb}, the tail latency is always under 150 ms, which is below common values of playback buffers used today [59].

Fall-back and recovery latency. Figure 10 measures the true latency to detect multicast link failure (left) and recovery (right), depending on D_{fb}. We notice that the proper time to detect that

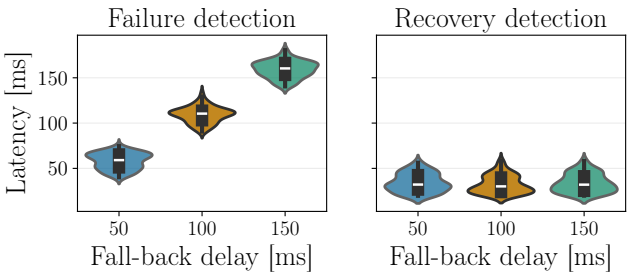

Figure 10: Latency of multicast link failure (left) and recovery detection (right) for $D_{fb} = \{50, 100, 150\}$ ms.

some link is failing depends on D_{fb} and the RTT, which is expected since the scheduler reacts to the *lack of acknowledgment* from the receiver for data sent on the flexicast flow. However, the time to consider the flexicast flow back alive depends on the RTT since it takes up to one RTT between data transmission and acknowledgment from the receiver to consider this flow again.

7 RELATED WORK

Content Delivery Networks (CDNs) emerged as a solution for one-to-many communications due to the historical difficulties of deploying inter-domain multicast [23]. CDNs distribute data from the edge using unicast protocols like TCP and QUIC. Some CDNs use multicast overlays to distribute video traffic to their edge servers [41].

The IETF discussed several approaches to ease the deployment of multicast between domains: Automatic Multicast Tunneling (AMT) [10], AMT defines a control loop to create tunnels connecting multicast islands and relies on DRIAD [31] to find these tunnel endpoints. TreeDN [29] suggests using tree-based delivery networks leveraging AMT. FCQUIC currently targets intra-domain deployments. Our future work will explore how to combine FCQUIC with these techniques and test them on the global Internet. End System Multicast (ESM) [13] presents a multicast overlay architecture where endpoints can perform packet replication instead of relying on an IP Multicast network. ESM requires the collaboration of multiple trusted endpoints (i.e., receivers) to establish the overlay, an assumption we do not take in Flexicast QUIC.

Two IETF drafts have proposed to extend QUIC to support multicast [33, 48]. HTTP over multicast QUIC [48] suggests using HTTP3/QUIC over an IP multicast network. The multicast behavior is implemented using HTTP and not QUIC. This design does not modify the QUIC stack. To our knowledge, it has only been partially implemented in nghq [9]. Multicast extensions for QUIC [33] specify extensions to enable QUIC to use a multicast tree. These extensions focus on the source authentication problem to avoid spoofing from malicious clients. It suggests using the AMBI [34] method, which forwards multicast packet hashes either using a Merkle tree using multicast or on the per-client unicast connections. In a companion paper [28], the authors present MCQUIC, a deployment architecture using fan-out servers to handle unicast connections with each client and a one-to-many connection to distribute data to the receivers. There is an open-source implementation of MCQUIC [44] currently supporting a subset of the control

frames. Still, it does not implement the one-to-many communication nor the reliability and congestion control mechanisms. As a result, a complete comparison with Flexicast QUIC is not possible. The `FC*` frames from FCQUIC build upon the draft [33]. FCQUIC leverages and extends Multipath QUIC, simplifying the design and enabling seamless fallback to unicast when multicast trees fail. In contrast, MCQUIC extends regular QUIC. We benchmark Flexicast QUIC at high speed and show alternatives (e.g., using `sendmmsg`) in case the is no underlying multicast tree. While MCQUIC explores the source authentication problem, we leave it as a future work for Flexicast QUIC and focus instead on scalability, robustness, and evaluations of our implementation.

8 CONCLUSION AND FUTURE WORK

We propose Flexicast QUIC, an extension of Multipath QUIC supporting flexible multicast. Flexible multicast is the ability for a source to securely and efficiently distribute data to a set of receivers using multicast trees where and when they are available and unicast delivery otherwise.

Flexicast QUIC combines per-receiver unicast paths and a shared flexicast flow, which receivers consider as *another path with different encryption keys*. An FCQUIC source can distribute live streams to multiple receivers while offering security, reliability, and scalability. FCQUIC receivers can seamlessly fall back on unicast whenever the flexicast flow fails without supporting another protocol.

We benchmark our multithreaded implementation and show that an FCQUIC source can support 1000 receivers for an aggregated throughput of >80 Gbps, while QUIC saturates at ~20 Gbps. We also demonstrate that Flexicast QUIC receivers can seamlessly fall back to unicast when there are failures on the multicast tree.

Our future work includes leveraging Forward Erasure Correction to improve the reliability mechanism; design more efficient congestion and flow controllers; integrate source authentication methods; exposing multiple flexicast flows in parallel, e.g., to provide multiple video streams encoded at different bit-rates for heterogeneous receivers; evaluate Flexicast QUIC in other use-cases such as large file distribution, e.g., software updates and video game releases; deploy Flexicast QUIC in inter-domain multicast scenarios, leveraging AMT [10] and TreeDN [31]; handle dynamic groups with key rotation mechanisms such as [12]. We will also consider more advanced schedulers compared to Section 6.2, including heterogeneous receivers with varying RTT and loss conditions. Such schedulers will also handle bottleneck receivers from impacting the remainder of the flexicast flow and adapt to the group size and per-receiver conditions (e.g., the *ack delay* and D_{fb}).

ARTIFACTS

We release the source code of Flexicast QUIC based on Cloudflare *quiche* and all the installation and evaluation scripts used for its evaluation at https://github.com/IPNetworkingLab/flexicast-quic. We encourage application developers to explore how to integrate FCQUIC and hope this will encourage more multicast deployment in the global Internet.

ACKNOWLEDGMENTS

Louis Navarre is an F.R.S-FNRS Research Fellow. This work has been partially supported by the Walloon Region as part of the funding of the FRFS-WEL-T strategic axis. We thank the SIGCOMM CCR reviewers for their valuable feedback.

REFERENCES

[1] 2023. FFmpeg. (2023). http://www.ffmpeg.org.
[2] 2023. GStreamer. (2023). https://gstreamer.freedesktop.org/.
[3] 2024. YouTube recommended upload encoding settings. (2024). https://support.google.com/youtube/answer/1722171?hl=en#zippy=%2Cvideo-codec-h%2Ccontainer-mp%2Caudio-codec-aac-lc%2Cbitrate.
[4] Akamai. 2022. Oops, we did it again. (2022). https://www.linkedin.com/pulse/oops-we-did-again-akamai-technologies.
[5] Kevin C Almeroth. 2000. The evolution of multicast: From the MBone to interdomain multicast to Internet2 deployment. *IEEE Network* 14, 1 (2000), 10–20.
[6] Aygün Baltaci, Hendrik Cech, Nitinder Mohan, Fabien Geyer, Vaibhav Bajpai, Jörg Ott, and Dominic Schupke. 2022. Analyzing real-time video delivery over cellular networks for remote piloting aerial vehicles. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 98–112.
[7] Tom Barbette. 2024. Poster: NPF: orchestrate and reproduce network experiments. In *2024 ACM Conference on Reproducibility and Replicability*.
[8] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 5–16.
[9] BBC. 2020. nghq: An implementation of Multicast QUIC. (2020). https://github.com/bbc/nghq.
[10] Gregory Bumgardner. 2015. Automatic Multicast Tunneling. RFC 7450. (Feb. 2015). https://doi.org/10.17487/RFC7450
[11] Bradley Cain, Dr. Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. 2002. Internet Group Management Protocol, Version 3. RFC 3376. (Oct. 2002). https://doi.org/10.17487/RFC3376
[12] Germano Caronni, K Waldvogel, Dan Sun, and Bernhard Plattner. 1998. Efficient security for large and dynamic multicast groups. In *Proceedings Seventh IEEE International Workshop on Enabling Technologies: Infrastucture for Collaborative Enterprises (WET ICE'98)(Cat. No. 98TB100253)*. IEEE, 376–383.
[13] Yang-hua Chu, Sanjay G Rao, and Hui Zhang. 2000. A case for end system multicast (keynote address). *ACM SIGMETRICS Performance Evaluation Review* 28, 1 (2000), 1–12.
[14] Cisco. 2008. Trading Floor Architecture. (2008). https://www.cisco.com/c/en/us/td/dEstringocs/solutions/Verticals/Trading_Floor_Architecture-E.html
[15] Cloudflare. 2023. Quiche: savoury implementation of the QUIC transport protocol and HTTP/3. (2023). https://github.com/cloudflare/quiche.
[16] Quentin De Coninck. 2022. Multipath support with non-zero length Connection IDs. (2022). https://github.com/cloudflare/quiche/pull/1310.
[17] Luis Costa and Rolland Vida. 2004. Multicast Listener Discovery Version 2 (MLDv2) for IPv6. RFC 3810. (June 2004). https://doi.org/10.17487/RFC3810
[18] Quentin De Coninck. 2022. The packet number space debate in multipath quic. *ACM SIGCOMM Computer Communication Review* 52, 3 (2022), 2–9.
[19] Quentin De Coninck and Olivier Bonaventure. 2021. Multiflow QUIC: a generic multipath transport protocol. *IEEE Communications Magazine* 59, 5 (2021), 108–113.
[20] Dr. Steve E. Deering. 1989. Host extensions for IP multicasting. RFC 1112. (Aug. 1989). https://doi.org/10.17487/RFC1112
[21] Stephen Deering, Deborah L Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. 1996. The PIM architecture for wide-area multicast routing. *IEEE/ACM transactions on networking* 4, 2 (1996), 153–162.
[22] Stephen E Deering. 1988. Multicast routing in internetworks and extended LANs. In *Symposium proceedings on Communications architectures and protocols*. 55–64.
[23] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. 2000. Deployment issues for the IP multicast service and architecture. *IEEE network* 14, 1 (2000), 78–88.
[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.
[25] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. (Aug. 2022). https://doi.org/10.17487/RFC9293
[26] Hans Eriksson. 1994. Mbone: The multicast backbone. *Commun. ACM* 37, 8 (1994), 54–60.
[27] Linux Foundation. 2015. Data Plane Development Kit (DPDK). (2015). http://www.dpdk.org
[28] Max Franke, Jake Holland, and Stefan Schmid. 2023. MCQUIC–A Multicast Extension for QUIC. *arXiv preprint arXiv:2306.17669* (2023).

[29] Lenny Giuliano, Chris Lenart, and Rich Adam. 2024. *TreeDN- Tree-based CDNs for Live Streaming to Mass Audiences*. Internet-Draft draft-ietf-mops-treedn-07. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mops-treedn/07/ Work in Progress.

[30] Hugh Holbrook and Storigen Systems. 2006. Source-Specific Multicast for IP. RFC 4607. (Aug. 2006). https://doi.org/10.17487/RFC4607

[31] Jake Holland. 2020. DNS Reverse IP Automatic Multicast Tunneling (AMT) Discovery. RFC 8777. (April 2020). https://doi.org/10.17487/RFC8777

[32] Jake Holland. 2020. IP Multicast: Next steps to make it real. (2020). NANOG79, available from https://youtu.be/2aihLUb1elg?list=PLO8DR5ZGla8i-aVXtTFRZ6l7BRBvYdrkO.

[33] Jake Holland, Lucas Pardue, and Max Franke. 2024. *Multicast Extension for QUIC*. Internet-Draft draft-jholland-quic-multicast-05. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-jholland-quic-multicast/05/ WIP.

[34] Jake Holland and Kyle Rose. 2022. *Asymmetric Manifest Based Integrity*. Internet-Draft draft-ietf-mboned-ambi-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mboned-ambi/03/ Work in Progress.

[35] Jana Iyengar and Ian Swett. 2021. QUIC Loss Detection and Congestion Control. RFC 9002. (May 2021). https://doi.org/10.17487/RFC9002

[36] Jana Iyengar, Ian Swett, and Mirja Kühlewind. 2024. *QUIC Acknowledgment Frequency*. Internet-Draft draft-ietf-quic-ack-frequency-10. IETF. https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency/10/ WIP.

[37] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. (May 2021). https://doi.org/10.17487/RFC9000

[38] Benedikt Jaeger, Johannes Zirngibl, Marcel Kempf, Kevin Ploch, and Georg Carle. 2023. QUIC on the Highway: Evaluating Performance on High-rate Links. In *2023 IFIP Networking Conference (IFIP Networking)*. IEEE, 1–9.

[39] Marcel Kempf, Benedikt Jaeger, Johannes Zirngibl, Kevin Ploch, and Georg Carle. 2024. QUIC on the Fast Lane: Extending Performance Evaluations on High-rate Links. *Computer Communications* 223 (2024), 90–100.

[40] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[41] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, et al. 2022. Livenet: a low-latency video transport network for large-scale live streaming. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 812–825.

[42] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kühlewind. 2024. *Multipath Extension for QUIC*. Internet-Draft draft-ietf-quic-multipath-11. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/11/ Work in Progress.

[43] Julien Maisonneuve, Muriel Deschanel, Juergen Heiles, Wei Li, Hong Liu, Randy Sharpe, and Yiyan Wu. 2009. An overview of IPTV standards development. *IEEE Transactions on broadcasting* 55, 2 (2009), 315–328.

[44] aioquic Max Franke, Jake Holland. 2023. MCQUIC implementation. (2023). https://github.com/MaxF12/aioquic.

[45] Steven McCanne and Van Jacobson. 1995. vic: A flexible framework for packet video. In *Proceedings of the third ACM international conference on Multimedia*. 511–522.

[46] Marco Mellia and Michela Meo. 2010. Measurement of IPTV traffic from an operative network. *European Transactions on Telecommunications* 21, 4 (2010), 324–336.

[47] Microsoft. 2022. Use multicast to deploy Windows over the network with Configuration Manager. (2022). https://learn.microsoft.com/en-us/mem/configmgr/osd/deploy-use/use-multicast-to-deploy-windows-over-the-network

[48] Lucas Pardue, Richard Bradbury, and Sam Hurst. 2022. *Hypertext Transfer Protocol (HTTP) over multicast QUIC*. Internet-Draft draft-pardue-quic-http-mcast-11. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-pardue-quic-http-mcast/11/ Work in Progress.

[49] Sanjoy Paul, Krishan K. Sabnani, JC-H Lin, and Supratik Bhattacharyya. 1997. Reliable multicast transport protocol (RMTP). *IEEE journal on selected areas in communications* 15, 3 (1997), 407–421.

[50] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. (Aug. 2018). https://doi.org/10.17487/RFC8446

[51] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. 2003. RTP: A Transport Protocol for Real-Time Applications. RFC 3550. (July 2003). https://doi.org/10.17487/RFC3550

[52] Tokio. 2024. Tokio: An asynchronous Rust runtime. (2024). https://tokio.rs.

[53] Nikita Tyunyayev and al. 2022. A high-speed QUIC implementation. In *Proceedings of the 3rd International CoNEXT Student Workshop*. 20–22.

[54] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[55] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. 2003. Overview of the H. 264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology* 13, 7 (2003), 560–576.

[56] IJsbrand Wijnands, Eric C. Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. 2017. Multicast Using Bit Index Explicit Replication (BIER). RFC 8279. (Nov. 2017). https://doi.org/10.17487/RFC8279

[57] Johannes Zirngibl, Philippe Buschmann, and et al. 2021. It's over 9000: analyzing early QUIC deployments with the standardization on the horizon. In *Proceedings of the 21st ACM Internet Measurement Conference*. 261–275.

[58] Johannes Zirngibl, Florian Gebauer, and al. 2024. QUIC Hunter: Finding QUIC Deployments and Identifying Server Libraries Across the Internet. In *International Conference on Passive and Active Network Measurement*. Springer, 273–290.

[59] Zoom. 2023. Zoom support: accessing meeting and phone statistics. (2023). https://support.zoom.us/hc/en-us/articles/202920719-Accessing-meeting-and-phone-statistics.