Scaling Flexicast QUIC for Large-Scale Software Distribution

Louis Navarre and Olivier Bonaventure, *Member, IEEE*

***Abstract*—Large-file distributions, such as video game patches and OS updates, generate traffic spikes, forcing ISPs to over-provision their networks beyond daily requirements. Content Delivery Networks usually deliver these files using unicast, generating large amounts of duplicate traffic. We reconsider multicast by building upon Flexicast QUIC (FCQUIC), a QUIC extension that combines efficient multicast delivery with robust per-receiver unicast paths, offering fallback when needed. We address two challenges related to large-scale multicast file transfer. First, the ACK implosion problem: as the number of receivers grows, acknowledgment traffic can overwhelm the source. We derive an adaptive acknowledgment strategy in which the source dynamically advertises an acknowledgment delay, capping the aggregate rate from the receivers below a defined threshold. Second, loose receiver synchronization: because receivers request the same file in concentrated but not simultaneous bursts, we design a file-rolling system which splits the file into blocks and sends them in a carousel, allowing late receivers to begin processing the application at the next block rather than waiting for a complete retransmission. We implement both mechanisms in FCQUIC and evaluate them on CloudLab with up to 1000 receivers. Our adaptive strategy caps the acknowledgment rate without degrading per-receiver goodput and outperforms both QUIC and NORM, a multicast state-of-the-art protocol, with numerous receivers across numerous scenarios, including heterogeneous receiver arrival and packet loss. We further model the CPU load of the FCQUIC source and show that the dominant factor is the acknowledgment delay, which we can control, allowing us to consider larger-scale deployments.**

***Index Terms*—Flexicast QUIC, Multicast, File transfer**

I. Introduction

Large-file distributions, such as video game releases and software updates, strain ISP networks. Many users regularly download files of tens to hundreds of GB. For example, the Fortnite game requires around 70-90 GB on PCs, and 35-45 GB on PlayStation 5 and Xbox Series [1]; new triple-A games usually weigh multiple tens of gigabytes [2]. Fortnite further distributes *seasonal* upgrades every ∼90 days, totaling around 10 GB each, and other popular games such as Rocket League and Valorant follow this model. League of Legends, with one of the largest active communities, applies an average of two patches per day [3]. Even if these patches are much lighter, they contribute to the large volume of video game updates, putting pressure on the ISP networks.

Organizations that distribute large files rely on Content Delivery Networks (CDNs). These CDNs emerged as solutions to improve scalability and enable the quick distribution of web content to millions of recipients. CDNs efficiently disseminate data within their networks and deploy edge servers in Service Provider (SP) networks or peer with them to reach receivers. Unfortunately, CDN edge servers rely on *unicast* protocols such as TCP [4] and QUIC [5] to send the data to these millions of receivers. Sending files via unicast is inefficient for large audiences because of the high volume of duplicate data traversing the network. As a result, the release of software updates significantly puts pressure on Internet Service Providers (ISPs) and Exchange Points (IXPs). For example, Fortnite's November 2, 2024, update induced a ∼100 Gbps increase (+20 %) in Roma at Namex [6]. The LONAP Ltd network reached ∼1.68 Tbps the same day, while the total traffic the following day at the same hour was only 1 Tbps. More recently, OpenReach, the largest UK access network, complained that another Fortnite patch pushed traffic to 372 Petabytes in a single day [7], breaking a new record. Next to these video game patches, system software updates also put pressure on network providers with new releases. Measurements from Apple iOS's September 2017 update showed a +483 % increase in CDN traffic peering with Apple's Meta-CDN shortly after the update release [8]. This study also found that this update caused congestion on unrelated peering links due to the large traffic spikes. More recently, similar events also caused sudden traffic spikes of +250 % due to Microsoft [9] and Apple [10] update releases.

In response, ISPs and IXPs must scale their networks to ingest these sporadic events [11], leading to an order-of-magnitude increase in network capacity beyond daily requirements [12]. Next to this, Content Delivery Networks address this issue by increasing the presence of their edge servers within large ISPs. For example, Akamai counts more than 4,400 points of presence spread over only ∼130 countries [13]. Applications also try new solutions to prevent saturating their customers' networks: in early November 2025, the release of the Call of Duty game began with a pre-download phase [14] that enabled users to download most of the game before finalizing it on release day, preventing traffic spikes. While a file transfer is not a one-to-many application at first glance, recent events indicate a concentration of requests for duplicate content within short periods [6]–[8], [10].

Multicast scalability over unicast. A more efficient way to distribute files is to rely on *multicast* protocols. With multicast, the source sends a single packet that the network replicates to reach all receivers. As such, *at most one packet crosses every link of the network*, minimizing the total bandwidth consumed. Multicast limits traffic in ISP networks by replicating packets only when needed, unlike unicast, which scales linearly with the number of receivers. Inter-domain multicast has historically been limited by deployment issues [15]. As a result, multicast file transfer is confined to enterprise networks [16], where operators have complete control over their networks and their hosts, or specific applications in research networks [17]. However, the industry is showing a rising interest in the use of multicast for large-scale communications for massive live-streaming [18], [19] and AI data-centers [20]. Multiple previous works attempted to disseminate files efficiently using multicast [21]–[24], leading to standardized protocols [25],

[26]. However, these protocols assume an end-to-end multicast network and do not provide unicast fallback when subsets of receivers lack such connectivity.

We reconsider the use of multicast to efficiently deliver software updates (or other large files) to large audiences. Given that CDNs deploy points of presence within ISPs to improve efficiency, we believe they could leverage their existing intra-domain multicast deployments to efficiently distribute such updates to large audiences. We build on our previous work, Flexicast QUIC (FCQUIC) as a reliable, secure, and flexible extension of QUIC [5], providing multicast when available and seamless unicast fallback otherwise. File distribution begins with the establishment of a regular QUIC connection and an HTTP/3 request from clients. If the server uses multicast to distribute the file, it upgrades the connection to use Flexicast QUIC. Thanks to the protocol's flexibility, clients that become a communication bottleneck or lack multicast support seamlessly fall back to unicast delivery without impacting the ongoing connection or the application.

The ACK implosion problem. Historically, large, reliable multicast communication has suffered from the *ACK implosion problem*, in which a single packet targeted to a large multicast group generates numerous acknowledgments, increasing packet processing overhead and limiting scalability. Multiple approaches have been proposed to mitigate the impact of ACK implosion, often adding complexity to aggregation mechanisms between end-hosts and sometimes involving intermediate nodes [27]. Recent multicast protocols use negative-acknowledgment (NACK)-based approaches to limit the scope of the problem [25], but they assume fully working multicast support, which is not always the case in real deployments.

With Flexicast QUIC, the source maintains per-receiver unicast paths during the communication and knows at any time the set of active receivers. Thanks to this feature, we design, implement, and evaluate an *adaptive delayed acknowledgment mechanism* in which the source dynamically adjusts the acknowledgment rate for its receivers by *pacing* their feedback. We also tackle the problem of heterogeneous arrival of receivers by designing a *file-rolling* system, in which the source splits the file into multiple blocks and repeatedly sends them in a *carousel* fashion, which allows late clients to process data as soon as the next incoming block is received.

Using the CloudLab [28] research platform, we evaluate our approach through emulation with up to 1,000 receivers, showing the benefits of multicast for large software distribution and the adaptability of our acknowledgment strategy. Because real scenarios involve many more receivers and CloudLab limits our experiments to $100\,\mathrm{Gbps}$, we study the factors that impact the scalability of a Flexicast QUIC server to determine whether it can sustain larger numbers of receivers. We observe that the main impact on the CPU load of our server is the *acknowledgment rate*, which we control with our adaptive strategy, and we show that our FCQUIC server could support up to 10,000 receivers with about 20 commodity threads on our experimental testbed. We analyze the impact of our adaptive acknowledgment strategy on congestion control and reliability mechanisms, highlighting trade-offs between scalability, achieved by reducing the ACK rate, and reactivity to congestion/loss events. We further explore the behavior of Flexicast QUIC under large software updates in various scenarios, including heterogeneous receiver arrival rates and packet losses, and compare it with NORM [25], the state-of-the-art approach for reliable multicast transfers.

Internet deployment. This paper focuses on providing a transport solution to distribute software updates at a large scale while limiting the multicast ACK implosion problem. For completeness, we briefly discuss why we believe that this approach works with current ISP networks. CDNs already have points of presence (PoP) with large ISPs to improve the efficiency of software distribution [29]. At the same time, ISPs usually deploy intra-domain multicast for specific use-cases such as IPTV [30], [31], but do not expose these multicast trees to CDN edges. We argue that the knowledge gained from Flexicast QUIC allows us to reconsider the availability of these servers to external entities, such as by providing ways to tariff the use of multicast by CDN sources since FCQUIC knows the exact set of receivers receiving the content.

Contributions. We make the following contributions.

- We design an adaptive acknowledgment strategy that allows the source to dynamically adjust the acknowledgment rate for its receivers. We mathematically derive conditions to limit the ACK implosion problem, and discuss the induced trade-offs.
- We propose the *file-rolling* system, in which the source splits the file into multiple blocks, allowing late receivers to process the next block and improving download completion by constant factors.
- We implement and evaluate our two systems in Flexicast QUIC, and emulate up to 1,000 receivers in CloudLab for up to $100\,\mathrm{Gbps}$ of multicast-aggregated traffic, and explore various scenarios, including different client arrival rates and heterogeneous losses.
- Because real deployments typically involve many more receivers, we analyze the scalability limits of our FCQUIC server and derive models to predict its CPU load, enabling extrapolation to larger receiver sets based on measurements in our testbed.

Artefacts. All the source code, experiment scripts, details to reproduce this work, and raw results are available on our public repository: https://github.com/louisna/fcquic-ack-experiments-ToN.

II. Background

QUIC [5] is a modern transport protocol aiming to replace TCP for the web. Among other features, QUIC provides stream multiplexing to address TCP's head-of-line blocking problem and embeds TLS 1.3 [32] in its design. Instead of the TCP four-tuple, QUIC identifies connections with Connection Identifiers (CIDs). Apart from the CID and a few fields, QUIC encrypts the header and payload of each packet. QUIC packets are *frame containers*: each packet is identified by a unique packet number and carries one or multiple frames, both for control and data. For example, `ACK` frames carry acknowledgments to the peer, while the `STREAM` frame carries reliable data. The multiple path management for QUIC [33]

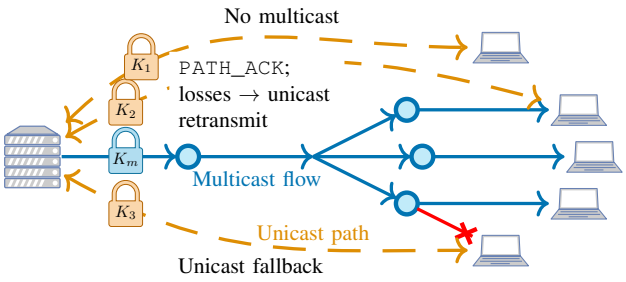

Fig. 1: Flexicast QUIC design. We omit the unicast paths of the two middle receivers for clarity.

(shortened Multipath QUIC) enables hosts to exchange data over different paths, e.g., cellular and Wi-Fi. To this end, it defines several control mechanisms to manage paths and path-specific acknowledgments through the PATH_ACK frame. The frame can be sent on one path while being acknowledged on another path. The extension defines individual congestion states for each additional path. Multipath QUIC is becoming an IETF standard track RFC.

Flexicast QUIC (FCQUIC) [34] is a recent research work extending Multipath QUIC. It enables a sender to reliably deliver data using unicast and/or multicast. Flexicast QUIC extends this multipath capability by 1) allowing more than two hosts to participate in a connection and 2) enabling the utilization of a shared unidirectional multicast flow[1] between a source and a set of receivers using a multicast distribution tree. A Flexicast QUIC connection starts with a standard QUIC handshake, during which the multipath and flexicast extensions are negotiated. Then, suppose the receiver requests content that can be delivered via multicast. The source sends an FC_ANNOUNCE control frame to this receiver containing the identifiers (R, the source IP address of the root of the multicast tree; G, the multicast destination address; the UDP port number; and the Connection ID) of the multicast flow that this receiver can join to receive this data more efficiently. Receivers create a new path (per the Multipath QUIC semantics) that will only be used to receive packets sent by the source using the (R, G) IP addresses: the *multicast flow*. The source can either use the multicast flow or all the bidirectional unicast paths to send data. Fig. 1 illustrates the design of Flexicast QUIC. From an efficiency viewpoint, the multicast flow is superior to sending the same data over all bidirectional paths. The bidirectional paths complement the multicast flow when some lost data needs to be retransmitted only to one or a few receivers, or to reach receivers that cannot currently receive multicast packets due to transient or permanent multicast problems.

In contrast to Multipath QUIC, which uses a single set of TLS keys shared between the two communicating hosts, FCQUIC uses separate sets of TLS keys for each flow. This brings several benefits from a security viewpoint. First, each receiver shares a set of TLS keys with the source to protect its bidirectional unicast path ($K_{\{1,2,3\}}$). All data and control frames exchanged over this path are only visible to the source and this receiver. The source also derives a set of TLS keys to encrypt and authenticate the data and control frames sent over the multicast flow (K_m). This set of keys is shared with each receiver by sending it in a control frame over that receiver's unicast path, encrypted with the receiver's individual key. Following Multipath QUIC's semantics, receivers send acknowledgments using the PATH_ACK frame on the unicast path, for packets received on the unicast path and on the multicast flow. Flexicast QUIC uses per-receiver acknowledgments and unicast congestion control algorithms to derive individual views of the congestion window for packets sent on the multicast flow. Then, the multicast flow congestion window is set to the minimum among all active receivers.

[1]Flexicast QUIC's semantics use the term *path* to refer to a *bidirectional* communication, and *flow* when it is unidirectional.

III. Scalable Acknowledgments

Reliable protocols rely on acknowledgments. In addition to the *data rate* from the sender to the receiver, acknowledgments add an *ack rate* in the reverse direction. QUIC receivers must acknowledge at most every two *ack eliciting* packets [5]. This mechanism works well in unicast because the *ack rate* is smaller than the *data rate*. In multicast scenarios, increasing the number of receivers inherently increases the *ack rate*, while the sender maintains the same transmission rate.

This phenomenon is commonly known as the *ack implosion* [35] problem, in which the bottleneck becomes the number of acknowledgments the sender must handle for every transmitted packet. Another strategy is to rely only on negative acknowledgments (i.e., report only the missing packets), but this approach is also subject to *nack implosion* if all receivers detect the same loss. NORM implements *nack echo/suppression* to limit this problem [25]. These mechanisms lack tracking the number of active multicast receivers, which is necessary to scale to large audiences, and must thus define methods independently of the group size. In this paper, we leverage the fact that a Flexicast QUIC source knows the exact set of active multicast receivers, to delay acknowledgments and dynamically adjust the delay. The idea of delaying acknowledgments, i.e., moving from acknowledging every (two) received packet, is already discussed for unicast within the IETF for TCP [36] and QUIC [37]. TCP endpoints must not delay acknowledgment by more than $500\,\mathrm{ms}$ [36]. The IETF draft on QUIC acknowledgment frequency recommends that endpoints send an ACK frame at least *once per round trip* to minimize impact on congestion and reliability mechanisms.

A. Estimating the acknowledgment rate

The initial FCQUIC paper explored the benefits of adding an acknowledgment delay in the initial design of FCQUIC, suggesting improved scalability because the source must process fewer packets [34]. The source statically advertised such acknowledgment delay during the announcement of the multicast flow through the FC_ANNOUNCE frame. However, this value is static and left to the application. Here, we extend Flexicast QUIC to dynamically adjust the expected acknowledgment rate from the receivers to the source. We define the *data rate* as $r(t)$, i.e., the rate in bps at time t at which the FCQUIC source

sends packets, including packet overhead. Table I summarizes the parameters we use in this paper. Then we call $n(t)$ the number of active receivers at time t that are receiving the multicast flow. The *ack rate*, $r_a(t)$ in bps, follows:

$$r_a(t) \propto r(t-\delta_t) \times n(t-\delta_t). \quad (1)$$

The δ_t in Equation 1 takes into consideration the fact that the acknowledgment rate at time t is a consequence of data sent earlier (i.e., δ_t before) to a group of a potentially different size at that time. For simplicity, we set $\delta_t = 0$, which holds at steady state when $n(t)$ and $r(t)$ are stable.

Symbol	Unit	Description	Page appearance
r	bps	Data rate	4
r_a	s	Acknowledgment rate	4
n	s	Number of active receivers	4
β	s	Acknowledgment delay	4
C	bps	Acknowledgment cap	4
α	–	Pareto shape	6

TABLE I: Description of the parameters.

Typical networks use an MTU of 1500 bytes. For bulk data transfers, QUIC packets containing `ACK` (or `PATH_ACK`) frames usually weigh ∼100 B. Furthermore, we assume that QUIC endpoints usually acknowledge every packet they receive. This is the default behavior in the QUIC stack we use, Cloudflare *quiche* [38]. We derive that

$$r_a(t) \approx \frac{1}{15} r(t) \times n(t) \quad (2)$$

B. Limiting the acknowledgment rate

We define C as the maximum expected acknowledgment rate summed over all receivers. C captures the maximum acknowledgment processing capacity of the server, which depends on the deployed environment. While we leave its estimation to network operators, we evaluate the sensitivity of our implementation through experiments for different values of C. As such, one must ensure that $r_a(t) \leq C$, which is currently dependent on the sending rate $r(t)$.

We suggest another strategy to limit $r_a(t)$ by dynamically advertising an *acknowledgment delay* to receivers. Instead of acknowledging every (at most) two ack-eliciting packets [5], this acknowledgment delay (let us call it $\beta(t)$) defines the time between two `PATH_ACK` frames, in seconds. To avoid clock synchronization, we uniformly and randomly select timers on each receiver to pace these frames, i.e., the time between two acknowledgment frames follows $X(t) \sim U(0, \beta(t))$, whose expectation is $\mathbb{E}[X(t)] = \frac{\beta(t)}{2}$. This gives the expected acknowledgment rate using the delayed strategy:

$$\mathbb{E}[r_a(t)] \approx \frac{800 \times n(t)}{\mathbb{E}[X(t)]} = \frac{1600 \times n(t)}{\beta(t)}, \quad (3)$$

again, assuming that each acknowledgment packet is ∼100 bytes (800 bits) long. Recalling that we cap the ack rate by C, we derive the value of β as defined by Equation 4.

$$\beta(t) = 1600 \frac{n(t)}{C}. \quad (4)$$

We only use a delayed-acknowledgment strategy when QUIC's default mechanism generates an ack rate above C. Putting it all together, we compute β as

$$\beta(t) = \begin{cases} 0, & \text{if } \frac{r(t) \times n(t)}{15} \leq C, \\ 1600 \frac{n(t)}{C}, & \text{otherwise.} \end{cases} \quad (5)$$

Large ack delay, performance, and congestion control. Delaying acknowledgments can reduce the emitter's reactivity in the presence of losses since it must wait for acknowledgments, which are delayed by β, to detect losses. Similarly, congestion events can be detected more slowly. This consideration brings a trade-off between capping the ack rate $r_a(t)$ and ensuring good detection of congestion and loss events when $n(t)$ is large and/or C low. To preserve this reactivity, the acknowledgment delay must remain within a small multiple of the RTT, and this multiple must be chosen carefully. To remain conservative, we compute an ideal lower bound on C. For example, to ensure that β remains no higher than 2 times the RTT, C is tailored by Equation 6:

$$C \geq \frac{1600 \times n_{\max}}{2 \times RTT_{\max}}. \quad (6)$$

In this computation, $RTT_{\max}$ corresponds to the highest RTT among all receivers. We use $RTT_{\max}$ as it is more conservative, even though other receivers may have much lower RTTs. This $RTT_{\max}$ is estimated from packets sent on multicast flows and acknowledged by receivers via the `PATH_ACK` frames. In ideal environments with no packet losses, delayed acknowledgments should not affect congestion control and reliability mechanisms. Adding an acknowledgment delay of β does not inflate the RTT samples from the source viewpoint, since the `ACK` and `PATH_ACK` frames contain the *ACK Delay* field, indicating the amount of time the peer waited before sending the frame for the corresponding acknowledged data [5]. The source accounts for this quantity when adjusting its RTT samples.

Moreover, adding a delay introduces batched acknowledgments, which free space in the congestion window and allow the FCQUIC to send new packets in bursts rather than continuously. A large value of β can lead to inefficient link utilization, as the source alternates between (i) short bursts of packets when the source's sending window frees upon receiving coarse `PATH_ACK` frames containing large ranges of acknowledgments, and (ii) long idle periods, precisely waiting for these acknowledgments. This phenomenon is particularly pronounced in the absence of pacing. We highlight the impact of too large values of β on the performance in Section V-C.

Distributing the ack delay. We extend FCQUIC with a new `FC_ACK_DELAY` frame to let the source inform receivers of updated values of β. Alongside the new value of β in microseconds, the frame contains a monotonically increasing sequence number to track the updates. The source sends this frame on the multicast flow for efficiency. It is also possible to send it individually along the unicast paths, using a specific value for each receiver. Our prototype uses the first approach, as a single packet reaches all receivers. Receivers that fall back on unicast receive subsequent `FC_ACK_DELAY` frames through their unicast path. These frames are unreliable, and

receivers that miss one such frame stick to older β values until the next update frame is received. Because β depends on the current delivery rate $r(t)$, the source may update and advertise new values after sending new packets on the multicast flow. We set the minimum interval between two `FC_ACK_DELAY` frames to 100 ms, but evaluate its impact in Section V-B.

C. Handling heterogeneous receivers

Our previous reasoning, which infers the computation of $\beta(t)$ from equation 5, assumes that all receivers share a similar RTT to the source on the multicast flow. In heterogeneous environments, the source may encounter receivers with low and high RTT relative to the acknowledgment delay. When $RTT \ll \beta$, the receiver sends less than an acknowledgment every round trip, which does not respect QUIC's recommendation regarding delayed acknowledgment [37], as the congestion state for this specific receiver is degraded.

A per-receiver β_i would improve reactivity for low-RTT receivers but cannot guarantee that the aggregate rate stays below C (Equation 6). For simplicity and multicast efficiency, we use a single β and assess the impact in Section V. Indeed, computing per-receiver $\beta_i(t)$ means that the source must send the `FC_ACK_DELAY` frames individually using the unicast path to the receivers, which is much less efficient if done at high frequency. This represents a deliberate trade-off between ACK rate control and congestion-control reactivity, which we leave to future work.

IV. File Rolling: Enabling Loose Synchronization

In large-scale software dissemination, receivers do not request the same content at exactly the same time, unlike in live streaming, where synchronization among clients is required by design. Recent events [6]–[8], [10] indicate that clients' requests concentrate within short periods, and clients arriving later than others want to receive the application data from the beginning, without preventing earlier clients from advancing in their downloads. We call this the multicast *loose synchronization* problem. Multicast protocols such as NORM [25] and FLUTE [26] repeat sending the same file, allowing late receivers to start from the beginning in the next iteration, though they must wait for the current one to finish, which adds time to complete the download. To expedite this process, we design a *file-rolling* approach, in which the source splits the file into blocks B_i and repeatedly sends each block over a separate FCQUIC stream. Receivers can join the multicast flow at any time. Their download starts as soon as a new FCQUIC stream (and thus a new block) begins.

HTTP/3 request and manifest response. The client initiates a standard QUIC handshake and then sends an HTTP/3 request for a specific object to the source. If the source wants to deliver this object using multicast, it responds with a manifest describing the file's chunking. Otherwise, it can deliver a standard HTTP/3 response using unicast. Fig. 2 illustrates a file transfer where we split the file into three blocks, B_1, B_2, and B_3. The manifest contains the stream identifiers used by the source to send the different blocks. The source repeatedly sends the blocks as receivers may arrive in any order (e.g., receiver R_3 in Fig. 2). The source subsequently cycles over the blocks using consecutive stream identifiers. For example, B_1 is first sent in stream s_1; B_2 in s_2 and B_3 in s_3, B_1 in s_4,.... QUIC's streams are identified by up to 62 bits, with the two lowest bits indicating the stream type [5], making stream ID exhaustion a non-concern. In Appendix A, we detail the exact mapping between the blocks and QUIC streams.

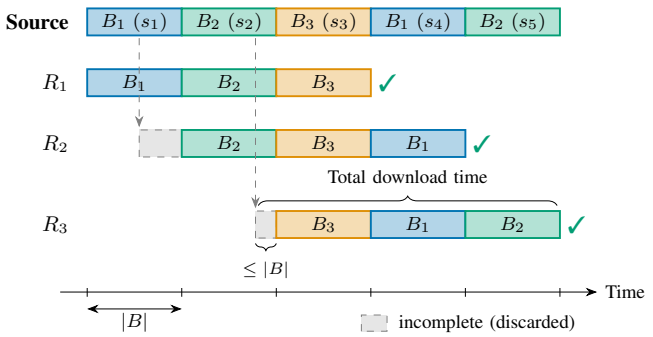

Fig. 2: File-rolling example with a file split into three blocks (B_1, B_2, B_3). R_1 joins at the start and receives all blocks in order. R_2 and R_3 join mid-block: the incomplete block is discarded, and reception resumes at the next block boundary. Each block is repeated in consecutive QUIC streams ($s_1 \ldots s_5$).

Receiving blocks. Receivers process blocks independently. In our example, the receiver R_2 processes B_2 before B_1, which will be received later. The application writes the content of B_2 to disk at its specific offset, using the manifest. As soon as the receiver completes all blocks (independently of the order of receptions), it closes the connection. Here, we assume that applications process the software update only after receiving the entire file, i.e., there is no advantage to receiving B_1 first rather than the other blocks.

Choosing the block size. If the file is not chunked into enough blocks, late receivers may need to wait longer before they can process valuable data. In Fig. 2, receiver R_3 cannot make use of the remaining packets from stream s_2 since it did not receive the first bytes, and must wait for the first bytes of s_3. We analyze the impact of the block size in Section V-E by implementing the file-rolling system in FCQUIC.

V. Experimental Evaluation

The initial FCQUIC paper [34] provided a first open-source implementation [39], based on a previous specification of the multiple path management for QUIC [40]. We update this implementation to build on the latest version of the IETF draft [33] and to simplify the source code. We further add our *ack delay* strategy and our *file-rolling* system on top, and evaluate these mechanisms through medium-scale emulation on the CloudLab infrastructure [28]. We compare unicast QUIC with regular Flexicast QUIC as we deliver the same file to an increasing number of receivers. First, we underscore the benefits of FCQUIC regarding network utilization (§V-A). We assess how our ack delay strategy dynamically caps the acknowledgment rate based on the number of active receivers without compromising the transmission rate (§V-B). Then, we infer a model estimating the CPU load of the Flexicast QUIC

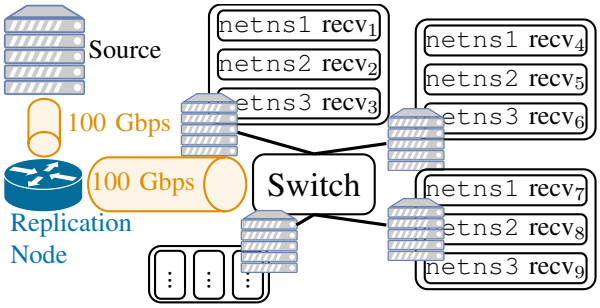

Fig. 3: Example of CloudLab topology used for our experiments, with three receivers per machine emulating receivers. We use network namespaces to isolate receivers.

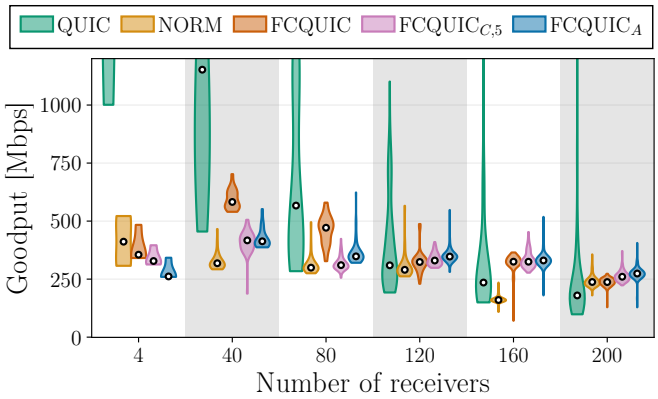

Fig. 4: Application goodput in our CloudLab setup. FCQUIC outperforms (unicast) QUIC with numerous receivers. Dots show the median value.

server to theoretically extrapolate to larger sets of receivers (§V-C). Finally, we explore the performance of FCQUIC across various scenarios, including different client arrival rates and block sizes, and under heterogeneous loss conditions, and compare our approach with NORM [25], a state-of-the-art multicast transport protocol.

CloudLab setup. We leverage CloudLab [28] to run a medium-scale multicast network, as illustrated in Fig. 3. We use six `d6515` servers. These servers are 16-core AMD EPYC 7452 at 2.4 GHz with 128 GB RAM. They are equipped with two 100 Gbps NICs. One server is used as the source. The *Replication Node* emulates an entire multicast network and performs software-based multicast replication using DPDK [41], reusing the same FastClick-based implementation as in our previous work [34]. Four servers emulate the receivers. Each host can support up to 250 receivers across different network namespaces, for a total of up to 1000 receivers simultaneously. All receivers are attached to the same LAN. As such, the cumulated replicated multicast traffic is capped to the 100 Gbps link speed.

Receiver arrival behavior. Recent software dissemination events [6]–[8], [10] show a concentration of requests for the same file near the software's release, with a long-tail distribution over time. We model this behavior using a *Pareto distribution*, a power-law model for event bursts with long tails. The Pareto distribution is characterized by two parameters: x_m, the scale parameter (the minimum arrival time), and α, the shape parameter, which defines the peak concentration and the tail width. We do not use this distribution to model inter-arrival times between clients. Instead, we sample the Pareto distribution to determine arrival times in seconds, so that the overall client's arrivals follow the Pareto curve. The client's arrival time probability is given by Equation 7:

$$P(X > x) \sim (x_m/x)^{\alpha}, x \geq x_m \tag{7}$$

The value of x_m has little impact, as it only determines the minimum time before the start of the experiment and the *first request* without impacting subsequent arrivals; we keep $x_m =$ 2 seconds for the remainder of this Section. Except stated otherwise, we use $\alpha = 1.5$, which is in the common range of values. With $\alpha = 1.5$, we expect 90 % of the receivers to request the file within the first 20 seconds of the experiment.

Measuring the application goodput. We report the *application goodput* as it is usually the most direct metric users identify when downloading files. We measure goodput as the ratio of file size to download time, in Mbps.

Compared protocols. We compare (unicast) QUIC (Cloudflare *quiche* version 0.23.4 [38]), NORM (version 1.5.9 [42]), and our new implementation of FCQUIC. For FCQUIC, we analyze three variants: 1) FCQUIC: using the standard QUIC acknowledgment mechanism; 2) FCQUIC$_{C,5}$: using a constant *ack delay* of 5 ms; 3) FCQUIC$_A$: using our adaptive acknowledgment strategy from Section III.

The NACK-Oriented Reliable Multicast (NORM) protocol is the closest related work to FCQUIC to deliver files using multicast. NORM focuses on scalability by relying only on negative acknowledgments (NACKs) and on multicast to distribute data to receivers. To avoid the (N)ACK implosion problem, the NORM source retransmits the first NACK message for a missing packet via multicast, thereby preventing other receivers from sending additional NACKs for the same packet. Receivers send these NACK messages via unicast, using the source address of the multicast packets. The source always retransmits losses on the multicast channel. To improve reliability, NORM caches the latest sent bytes in case late retransmissions are needed. The exact cache size is determined by the application. We use 100 MB of cache during all experiments, meaning that the source can retransmit up to 100 MB older data to late receivers. Because NORM does not use per-receiver unicast connections, receivers without (transient) multicast connectivity cannot receive the content. Therefore, we assume end-to-end multicast connectivity.

A. Performance in a perfect environment

We start by measuring the goodput offered by QUIC, NORM, and the three variants of FCQUIC as the number of receivers increases, in a perfect environment. Fig. 4 shows the per-receiver goodput and Fig. 5 the throughput measured on the Replication Node. Except stated otherwise, we use a maximum acknowledgment rate cap of C =100 Mbps and a

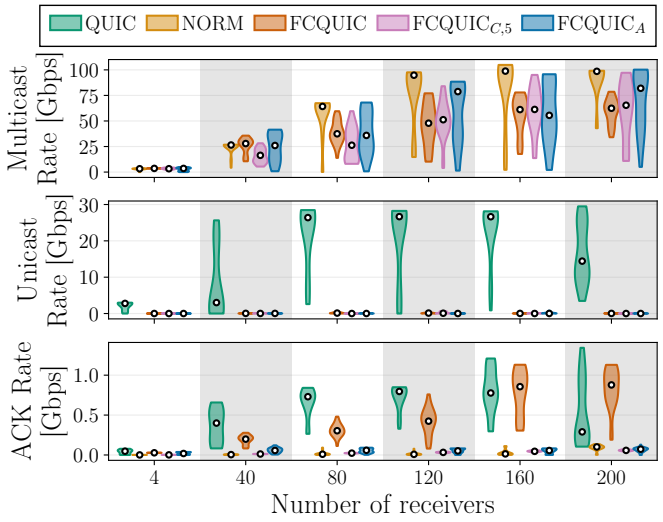

Fig. 5: Throughput measured on the Replication Node during the experiments from Section V-A: Multicast (top), unicast (middle), and ack (bottom) rates.

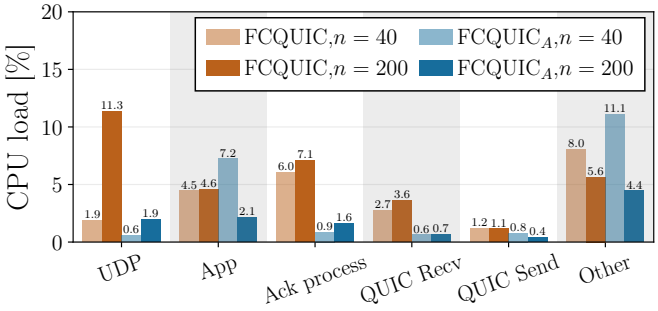

Fig. 6: Contributors of the CPU load on the FCQUIC source, comparing standard and our adaptive acknowledgment strategy, for 40 and 200 receivers. We only report elements that vary significantly with n. *App* includes application-specific tasks, such as reading data from a disk file. *QUIC Send* typically involves tasks such as sending packets on the multicast flow and per-receiver retransmissions. We used the `perf` tooling suite to collect these samples.

block size B =100 MB. We analyze the impact of these two parameters in Sections V-B and V-E.

FCQUIC provides higher goodput as the number of receivers increases. Unicast QUIC outperforms multicast protocols when a few receivers request the same file, since the flows do not saturate the 100 Gbps bandwidth limit. The per-receiver goodput matches previous results on the evaluation of the Cloudflare *quiche* stack [43]. This value decreases linearly with the number of receivers, reaching a median of less than 200 Mbps at 200 receivers. The measured unicast throughput (including protocol overhead) in Fig. 5 indicates that the server reaches a peak of 30 Gbps. As in previous results [34], the server CPU load reaches full utilization, which becomes the bottleneck before the bandwidth limit. We observed CPU contention on the servers emulating clients, so QUIC clients processed packets in batches before sending a single `ACK` frame, reducing the ACK rate.

The per-receiver goodput remains within the 250-500 Mbps range, independent of the number of receivers, for Flexicast QUIC, across the three acknowledgment strategies. For fewer receivers, we observe noticeable variance, but the goodput converges as we deliver the data to more clients simultaneously. For $n \geq 120$ receivers, FCQUIC provides higher goodput than QUIC. Fig. 5 shows that FCQUIC linearly increases the cumulated multicast throughput until it reaches ∼100 Gbps for 200 receivers, explaining the better results compared to QUIC, underlining the benefits of multicast replication.

In a perfect environment, NORM provides similar goodput to FCQUIC. Except for 4 receivers, FCQUIC variants even consistently provide higher goodput than NORM. We explain this difference by the lack of a *file-rolling* system for NORM: late receivers either wait for the next transmission of the file from the NORM source or request multicast retransmission from the cache, thereby slowing other receivers. This lack of a rolling mechanism leads to performance degradation, as seen in Fig. 4, where the arrival of late receivers prevents the source from sending new data. The link utilization of NORM is systematically higher than that of FCQUIC, despite the lower per-receiver goodput. We observed significantly higher drops from the software router with FCQUIC$_A$ than with NORM: for 200 receivers, the router dropped 14,974 FCQUIC packets but only 538 NORM packets. We explain this gap by a limitation of our experimental setup: the *Replication Node* struggles to handle both multicast and acknowledgment traffic at high speed on a single node.

Our adaptive acknowledgment strategy for FCQUIC induces a lower acknowledgment rate without impacting the goodput. Although the difference is not significant, both Figures show a positive impact of the adaptive strategy (FCQUIC$_A$, using a maximum rate of C =100 Mbps) over the standard procedure that acknowledges all packets (FCQUIC): the acknowledgment rate is significantly lower (median of ∼80 Mbps) compared to the standard approach (∼900 Mbps). Because the source constantly adjusts the acknowledgment delay β, it correctly caps the cumulated rate from the clients to the server. FCQUIC$_{C,5}$ similarly limits the acknowledgment rate by imposing a constant value of $\beta = 5$ milliseconds, which lacks dynamics to the number of active receivers.

Our adaptive strategy lowers the CPU load on the FCQUIC source. In Fig. 6, we report the main contributors of the CPU load on the source using the standard and our adaptive acknowledgment strategy, measured using `perf` during the experiments. We exclude memory copy operations (`memmove`) and the overhead of the `tokio` runtime, which we discuss later. The Figure illustrates the positive impact of delaying acknowledgments: the source must process fewer QUIC packets (*QUIC Recv*), and spends less time aggregating acknowledgments from receivers (*Ack process*). We note a side effect of handling 200 active connections: the *UDP lookup* is performed by the kernel, which must poll all active socket descriptors for incoming packets. With FCQUIC$_A$, the kernel polls these sockets less frequently, reducing the CPU load from 11.3 % to 1.94 % for $n = 200$ receivers. Because FCQUIC$_A$ spends less time processing packets and acknowledgments, we

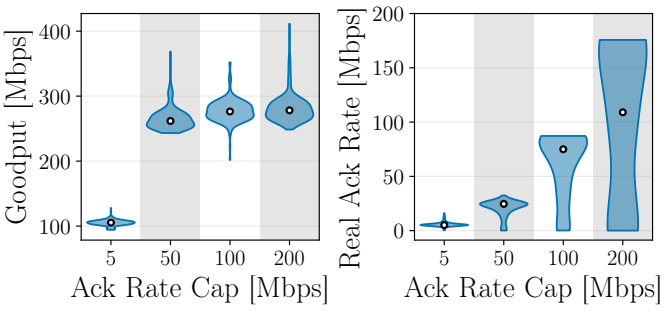

Fig. 7: Per-receiver goodput and real measured acknowledgment rate (r_a) when varying the acknowledgment rate cap (C), using our adaptive strategy FCQUIC$_A$.

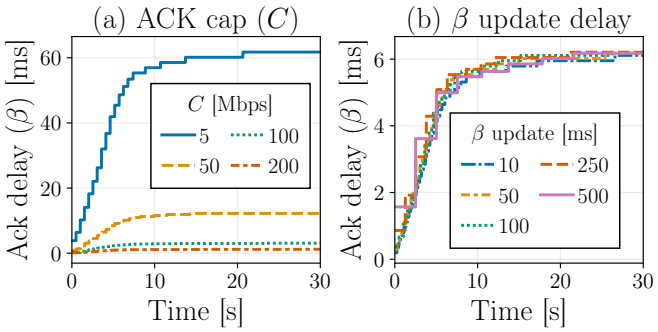

Fig. 8: The acknowledgment delay β adjusts to the maximum cap C (left); increasing the β update frequency adjusts more smoothly the acknowledgment rate for receivers (right), though the per-receiver goodput did not get affected for the tested ranges of frequency update.

notice that memory copies accounted for 50 % with FCQUIC$_A$ instead of 12 % with the standard acknowledgment procedure, showing the benefits of FCQUIC$_A$.

We do not show the `tokio` runtime overhead in the Figure for clarity, but it accounts for the majority of the CPU load. For FCQUIC$_A$ and 200 receivers, swapping and waking tasks across threads accounted for more than 50 % of the CPU load, but only 12.25 % with FCQUIC for the same number of receivers. This result explains that 1) because the FCQUIC$_A$ source processes fewer packets, tasks are more often in an idle state, and `tokio` must wake them up more regularly; 2) `tokio` is convenient but not optimized for high-intensity packet processing. We note that the CPU load percentage of `tokio` is higher for FCQUIC$_A$ than for FCQUIC because other elements, such as UDP poll, require fewer cycles.

Takeaway. In a perfect environment, FCQUIC and NORM achieve higher goodput than QUIC because unicast saturates the server's CPU. Thanks to our *file-rolling* system, FCQUIC even outperforms NORM. The adaptive acknowledgment strategy does not affect per-receiver goodput while limiting the *ACK rate* and reducing the source's CPU impact because it must process fewer packets. Our experimental testbed limits the link utilization of FCQUIC relative to NORM because the Replication Node struggles to handle multicast traffic in one direction and unicast acknowledgments in the other.

B. Dynamic acknowledgment behavior

We now assess how the source dynamically adjusts the acknowledgment delay, β, subject to a fixed maximum acknowledgment cap, C. We run the same experiment with $n = 200$ receivers and $C = \{5, 50, 100, 200\}$ Mbps. Fig. 7 summarizes the per-receiver goodput and the real acknowledgment rate r_a measured on the Replication Node. The right sub-figure shows that the measured r_a adjusts to the maximum expected C, thereby validating our approach. We observe a similar per-receiver goodput when $C \geq 50$ Mbps, though it improves slightly with higher caps. However, we see degradation for low acknowledgment caps, which is explained by the induced acknowledgment delay β relative to the RTT. For instance, when $C = 5$ Mbps, $\beta \approx 64$ ms, while the RTT is approximately 2 ms, yielding $\beta \approx 30 \times$ RTT.

Such a large value of β yields batched acknowledgments, received in bursts. As discussed in Section III, this effect leads to alternation between bursts of packet transmission from the FCQUIC source until the congestion window is filled, and long idle periods waiting for acknowledgments to free space in this congestion window. As such, the source under-utilizes the link, and the global goodput decreases. In our implementation, this effect is reinforced by the occasional packet drop on the Replication Node due to software-based replication.

The left part of Fig. 8 shows the evolution of β as receivers join for the same values of C, confirming that β correctly adjusts to incoming receivers and the maximum acknowledgment rate cap. Until now, we set that Flexicast QUIC distributes the updated β values through the `FC_ACK_DELAY` frames every 100 ms. The right-hand side of Fig. 8 further shows the evolution of β as we change this default delay of 100 ms. The global shape of the evolution of β is independent of the update delay. However, as expected, the granularity decreases with increasing delay. We did not observe a significantly higher acknowledgment rate r_a when updates were less frequent; we attribute this lack of difference to the precision of the acknowledgment rate estimation on the Replication Node, which is measured as an average every second. To mark a noticeable impact caused by the update delay, multiple tens of receivers should connect to the server within a few (tens of) milliseconds. However, increasing the update frequency incurs overhead, as each `FC_ACK_DELAY` frame weighs approximately 8 bytes (depending on the value of β), leaving less room to send application data.

C. Scaling: Flexicast QUIC CPU consumption

In the previous Section, we limited the receiver set to 200 because the replicated multicast traffic fully utilized the 100 Gbps bandwidth limit of the servers from our testbed. In this Section, we go further and explore the limits of the Flexicast QUIC source with our adaptive acknowledgment strategy while we significantly increase the number of receivers. To this end, we measure the CPU load on the server and statistically estimate the factors impacting its evolution, showing that *the*

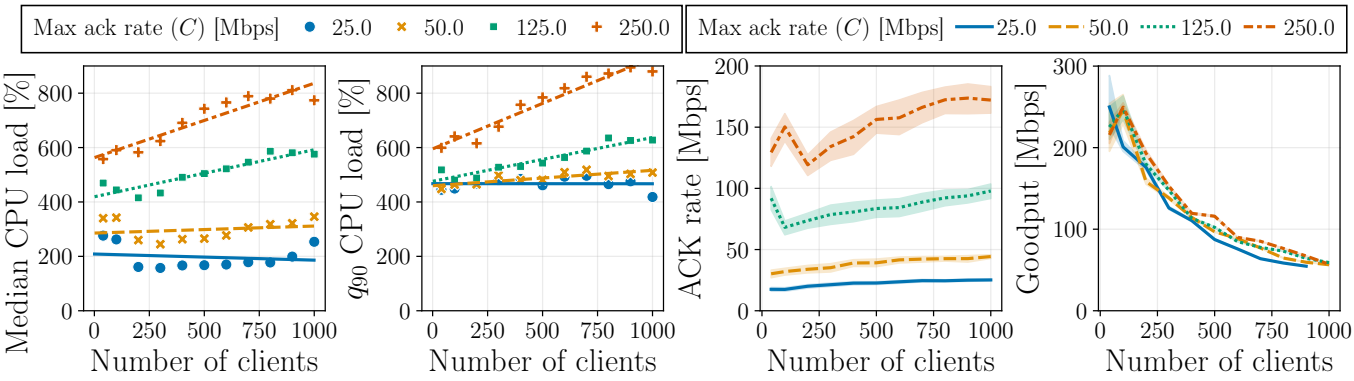

Fig. 9: Median and q_{90} of the CPU load when increasing the maximum acknowledgment rate (C) and the number of receivers (n) with FCQUIC. Scatters represent real samples from the CloudLab evaluation; the lines are computed using least-squares polynomial fits of order 1. The right sub-plots, respectively, report the actual ACK rate measured during our experiments and the per-receiver goodput, which decreases linearly as we saturate the 100 Gbps link for $n = 200$ receivers.

main influence stems from the acknowledgment rate, which we can dynamically control with our acknowledgment strategy.

We run the same experiments as in the previous Section, with up to $n = 1000$ receivers, in increments of 100 receivers, using FCQUIC$_A$ and multiple acknowledgment rate caps (C). Fig. 9 reports the results, respectively showing the median and q_{90} CPU load on the source, the measured ack rate r_a and the per-receiver goodput. The CPU load is summed over all active CPUs on the server, while the FCQUIC$_A$ source runs with 10 `tokio` workers (i.e., threads).

The source CPU load is mostly influenced by the acknowledgment rate, and slightly by the number of active receivers. The two left sub-figures highlight the impact of the acknowledgment rate on the server load. For the same number of receivers, increasing the acknowledgment rate cap C (i.e., vertical comparison) significantly increases the median and q_{90} load: for 1000 receivers, the median load ranges from ∼250 % with a 25 Mbps cap to ∼800 % with a 250 Mbps cap, with steady increase for intermediate caps. The 90^{th} percentile follows a typical shape, though the differences are less pronounced when considering only the high-load samples. The left part of Fig. 9 also shows that, for larger values of C, the CPU load increases with n, the number of active multicast receivers. Two reasons justify this result. First, even if we limit the per-receiver acknowledgment rate with FCQUIC$_A$, this mechanism only restricts the rate at which QUIC packets reach the source. However, the source still processes a wider range of acknowledged packets as the number of receivers increases to ensure full reliability; more receivers mean more processing of such ranges. Second, the third sub-figure shows the measured acknowledgment rate during these experiments. For $C \geq 125$, the ACK rate continues to increase with $200 \leq n \leq 1000$. In some prospects, increasing n also increases r_a, thereby indirectly affecting the CPU load.

The ACK rate and CPU load share a strong linear dependency, with a correlation score higher than 0.9. We quantitatively measure the impact of r_a and n on the CPU load using the *correlation*, as reported in Table II. The Table highlights that the linear dependence between CPU load and the acknowledgment rate is strong, with a correlation > 0.9 for both the median and q_{90} load. Still, as discussed previously, the number of active receivers, n, also affects the CPU load, especially for the q_{90} samples, with a correlation of 0.3235. For completeness, we also report the correlations between the CPU load and the multicast (r_{mc}) and unicast (r_{uc}) rates, although they are not the main contributors.

CPU Load	r_a	n	r_{mc}	r_{uc}
Median	**0.9622**	0.1719	−0.0557	0.2154
q_{90}	**0.9221**	0.3235	−0.2687	0.0018
q_{99}	**0.7789**	0.3783	−0.432	−0.1617

TABLE II: Correlation between the CPU load on the FCQUIC source and the acknowledgment rate (r_a), number of receivers (n), multicast rate (r_{mc}), and unicast rate (r_{uc}). The ack rate is the most significant factor affecting the CPU load on the source, both at the median and the 90^{th} percentile.

Modeling the CPU load. We compute a least-squares linear regression using r_a and n to model the CPU load of the FCQUIC$_A$ source, allowing us to estimate the server's CPU load at a larger scale than what we can achieve with CloudLab. Using standard models from `scikit-learn`, we derive the median and q_{90} CPU load estimations based on the collected samples from $100 \leq n \leq 1000$: $\widehat{\text{CPU}}_{50}$ and $\widehat{\text{CPU}}_{90}$. The acknowledgment rate r_a is difficult to estimate in advance and, by construction, $r_a \leq C$. Moreover, the third sub-figure in Fig. 9 confirms that $r_a \leq C$, allowing use to estimate:

$$\begin{aligned} \widehat{\text{CPU}}_{50}(C,n) &\approx 3.48 \times C + 0.0475 \times n + D_1, \\ \widehat{\text{CPU}}_{90}(C,n) &\approx 2.14 \times C + 0.0981 \times n + D_2. \end{aligned} \tag{8}$$

where C is expressed in Mbps for readability. The constants D_1 and D_2 stem from the constant CPU cost of the FCQUIC server, which is independent of the number of receivers and acknowledgment rate, e.g., generating and sending packets on the multicast flow. These two equations provide insight into *how* the CPU load of the FCQUIC$_A$ source evolves as the number of receivers increases, subject to acknowledgment capacity limitations. These equations model the FCQUIC$_A$ CPU

consumption under load, i.e., during active file transmission, based on samples collected during our measurements.

The *coefficient of determination* (R^2) represents the statistical dependence of the variance of one (set of) variable(s) explaining the variation of another through the linear regression model. In a nutshell, this coefficient measures *how well* the variables explain the evolution of another variable, ranging from 0 to 1. We measured $R^2_{q_{50}} = 0.93$ and $R^2_{q_{90}} = 0.9$, justifying that r_a and n significantly contribute to the evolution of the CPU load. The exact coefficients from Equation 8 are specific to our testbed, though the tendency should be similar in other setups.

The 99th percentile CPU load is also impacted by the acknowledgment rate. Because each experiment collected ∼175 samples of CPU load, we focused on the median and q_{90} to avoid relying too much on potential outliers. Still, we report the q_{99} CPU load correlation factors in Table II, confirming that the main factor affecting it remains the acknowledgment rate, even at peak load. However, the $R^2_{q_{99}}$ coefficient of determination falls to 0.7. r_a and n remain the primary contributors to the q_{99} CPU load, but other factors also influence it. These peak-load events occur when numerous receivers occasionally synchronize their acknowledgments, and when the application reads large chunks of data from memory; we leave such investigation for future work.

Decreasing the acknowledgment rate too much negatively impacts the file download performance. From Equation 8, one can constantly decrease the acknowledgment rate cap C to balance with an ever larger receiver set n. This is theoretically possible, though it raises concerns regarding the data transmission rate r, as discussed in Section III. As an example, the right-end side of Fig. 9 shows that *none of the 1000* receivers completed the file download with $C = 25$ Mbps. From Equation 5, we compute a theoretical $\beta = 64$ ms; our experimental results report a maximum value of $\beta = 63.423$ ms in this experiment. In other words, receivers could only send acknowledgments every ∼64 ms to meet the 25 Mbps acknowledgment cap rate. With an RTT of ∼2 ms, it means that receivers send less than an acknowledgment every 30 RTT, far below the advised values [37]. In addition to poorer RTT estimation, ultimately impacting the congestion controller and data transmission rate, the reactivity of the source to congestion and loss events is also affected. As such, we reported a multicast send rate of only ∼53 Gbps with $C = 25$ Mbps; the rate increases to > 70 Gbps for higher acknowledgment caps.

In Section III, we stated that β should be at most equal to a few multiples of the RTT. With 1000 receivers sharing a similar RTT of 2 ms and ensuring that $\beta \leq 10 \times RTT$ yields $C \geq 80$ Mbps. From Fig. 9, we observe that even for $C \geq 50$ Mbps, the file download completes correctly despite receivers sending fewer than an acknowledgment packet each RTT. This observation confirms the aforementioned trade-off between reactivity to congestion/loss events and better link utilization (i.e., increasing C) and the limitation of acknowledgment processing (decreasing C), which depends on the application and environment in which operators deploy Flexicast QUIC.

Extrapolating to 10,000 receivers would require less than 20 working threads for the Flexicast QUIC source. We can take Equation 8 and tailor C (Equation 6) to estimate the CPU load for 10,000 receivers. For example, if we assume a 10 ms RTT between the source and each receiver, and, to be conservative, state that $\beta \leq 5 \times RTT$, we obtain $C \geq 10{,}000 \times \frac{1600}{0.01 \times 5} = 320$ Mbps. Then, $\hat{\text{CPU}}_{50} \approx 1589\ \% + D_1$; Similarly, $\hat{\text{CPU}}_{90} \approx 1666\ \% + D_2$, i.e., about 20 working threads should be required to handle the 10,000 receivers in that context, not accounting for the fixed cost of sending the packets on the multicast flow. We note that this estimate accounts only for CPU load and assumes a linear model at scale. At 10,000 receivers, the memory footprint and the number of open socket descriptors may become bottlenecks.

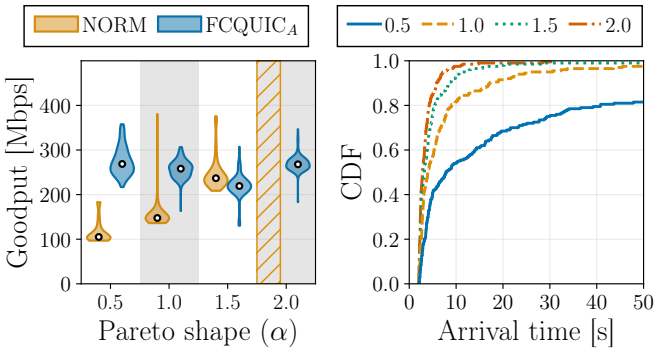

Fig. 10: Left: FCQUIC$_A$ provides a steady per-receiver goodput independently of the receiver arrival rate. Right: receiver arrival distribution for each α, with $n = 200$ receivers. For $\alpha = 2$, no NORM receiver completed the download.

Takeaway. The CPU load on the Flexicast QUIC source is heavily dependent on the acknowledgment rate r_a and the number of active receivers n. We establish that r_a is the main factor, directly influencing the median and q_{90} CPU load, while increasing the number of receivers, albeit slightly, also affects this load. By dynamically adjusting the acknowledgment rate, we can limit the server load. We derive equations to estimate the CPU load for larger sets of receivers that we could not obtain via CloudLab, and show that an FCQUIC server can handle 10,000 receivers simultaneously with ≤ 20 threads without significantly impacting the congestion controller and reactivity to loss and congestion events.

D. Impact of the arrival rate

We now explore the impact of the parameters introduced in the experimental setup: the receiver arrival rate α from the Pareto model and the block size B of the file rolling system. We start by varying the receiver arrival rate, α, for $n = 200$ receivers and compare NORM to FCQUIC$_A$ in Fig. 10. The right-hand side of the Figure shows the distribution of arrival time of the 200 receivers as a function of α. Independent of the arrival rate, FCQUIC$_A$ provides a steady goodput around 250 Mbps. We attribute this robustness to the file-rolling system, which allows late receivers to process application data "in the middle" of the transfer of a block.

On the contrary, the goodput provided by NORM improves with α: when receivers arrive sporadically (smaller values

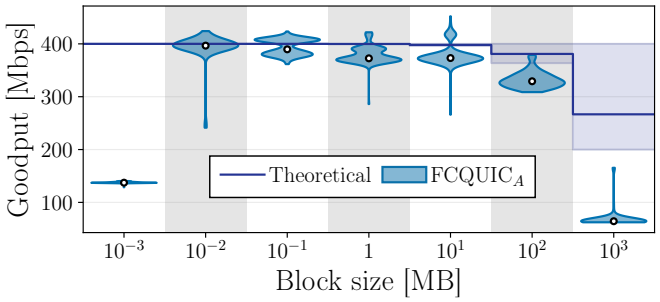

Fig. 11: Theoretical and measured goodput when varying the block size for $n = 200$ FCQUIC$_A$ receivers. Goodput increases with shorter blocks, whereas too low block sizes incur significant overhead.

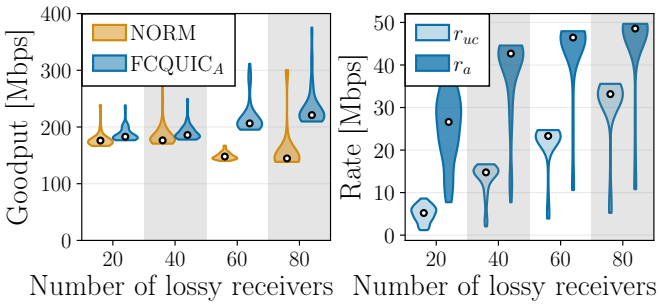

Fig. 12: Impact of heterogeneous random losses on the goodput of FCQUIC$_A$ and NORM (left), and the resulting unicast (r_{uc}) and acknowledgment rate (r_a) (right).

of α), NORM receivers must synchronize with the source, potentially during an ongoing file transmission. Every time a new receiver joins the group communication, it detects missing data (i.e., the beginning of the file) and sends NACK messages to the source. Upon receiving these NACKs, the source sends repair packets containing the missing data, provided it still has the missing data in cache. We set the source's default size to 100 MB. If the requested data is no longer in cache, the receiver must wait for the next file transmission. For example, with $\alpha = 0.5$, numerous receivers arrive after the source releases the initial file data from its cache; they thus need to wait for the next transmission of the file to process application data, which explains the lower goodput. Fig. 10 shows that some receivers reached a higher goodput when $\alpha = 0.5$. These receivers arrived earlier and completed the file download during the first transmission. However, we observe that their goodput remains below 200 Mbps due to numerous retransmissions requested by late receivers.

As the receiver arrival density increases, NORM's goodput improves because the probability that missing data is still in cache also increases for late receivers. We note that for $\alpha = 2$, no NORM receiver completed the file download. Here, almost all late receivers manage to join the group communication before the source drops the first application data from its cache. As a result, each new receiver repeatedly sends NACK messages to the source to request retransmissions, preventing the source from sending new data until all these late receivers join the group. After all receivers are synchronized, the source restarts sending fresh application data, resulting in lower goodput due to incorrect congestion-state estimation caused by late receivers. Ultimately, the source struggled to deliver the whole file within the experiment's lifetime.

Takeaway. NORM is dependent on the client's arrival rate. Worse, multicast retransmissions for late receivers degrade the goodput of initial receivers. On the contrary, our file-rolling system atop Flexicast QUIC allows late receivers to process application data sooner, and keeps the source from repeatedly retransmitting older data.

E. Impact of the block size

The file-rolling system chunks the file into blocks of specific sizes, allowing late receivers to process application data at the next block. As such, the block size impacts the goodput: the larger the block, the longer a late receiver might have to wait for the current block to complete before processing the next block from the start. We analyze this behavior with $n = 200$ receivers using FCQUIC$_A$. Fig. 11 reports the per-receiver goodput with different block sizes, as well as theoretical (and ideal) bounds on the goodput. This theoretical region is computed considering a constant transmission rate of 90 Gbps with no overhead, assuming that the receiver may join anytime between the start (i.e., it must wait for the entire transmission of the block) and the end (i.e., it can directly process the next block) of a block already being transmitted. FCQUIC$_A$ reaches this bound when the block size is 10 kB, but the goodput decreases faster as we increase the block size. This is mainly due to the source not maintaining a steady 90 Gbps bit rate when late receivers join with large blocks.

Smaller blocks improve the goodput, but induce overhead if too low. The theoretical curve shows that one can only expect to improve the goodput by decreasing the block size. We observe an improvement from 1 GB to 10 kB in our measurements, as receivers wait less time before processing application data. However, when the block size is 1000 B (i.e., each block fits in less than a single QUIC packet), the goodput shrinks to 130 Mbps. With such low block sizes, the protocol overhead becomes significant: QUIC endpoints must maintain state for numerous streams and consume numerous stream identifiers. This also raises flow-control issues, since QUIC endpoints must allow a large number of streams to be opened simultaneously. For example, with an RTT of 5 ms at a reception rate of 400 Mbps, at least 250 streams must be opened simultaneously. Moreover, 1000 B streams do not fill an entire QUIC packet, thus wasting room and further increasing the protocol overhead. Finally, the theoretical curve shows that, in our scenario, the potential gains from shortening the block size from 1 MB to 10 kB are negligible.

F. Impact of heterogeneous losses

Finally, we analyze the impact of losses on the goodput for both FCQUIC$_A$ and NORM in Fig. 12. We add 0.1 % random losses individually to an increasing number of receivers. The overall loss rate on the multicast tree is hence $\approx 1 - 0.999^l$ where l is the number of lossy receivers; for $l = 80$, the loss rate is ≈ 7.7 %. NORM uses Forward Error Correction

packets to recover from losses on distinct clients. Thanks to this approach, it minimizes the number of retransmitted packets sent on the multicast distribution tree. The per-receiver goodput decreases slightly from $l = 20$ to $l = 80$, due to the increasing number of retransmissions.

Flexicast QUIC provides similar robustness as NORM, e.g., when $l = 20$. The performance gap increases as the number of lossy receivers increases. We explain this difference by the particularity of the experimental setup. Because the replicated multicast traffic is carried through a single $100\,\mathrm{Gbps}$ link, each multicast retransmission from the NORM source, being replicated to all receivers, increases the utilization of the $100\,\mathrm{Gbps}$ link. As NORM uses FEC repair packets to recover from losses, a single repair can recover distinct losses across different clients. However, such repair may not be useful to all clients (e.g., lossless ones) but consumes more bandwidth than the per-receiver unicast retransmissions of FCQUIC.

The right-hand side of Fig. 12 shows the distribution of the unicast (r_{uc}) and acknowledgment rate (r_a) for FCQUIC$_A$. The unicast rate includes per-receiver retransmissions due to packet losses and increases linearly with l: for $l = 80$, only $35\,\mathrm{Mbps}$ of unicast traffic is needed to handle the loss events, suggesting that per-receiver retransmissions in this scenario are not more costly than multicast retransmissions. We expect that for heavier losses, multicast retransmissions (i.e., NORM's mechanism) may become more performant, but we do not expect the sender to maintain such a high transmission rate under these loss events. The Figure also reports the acknowledgment rate r_a, which increases with l because receivers send `PATH_ACK` frames when they detect gaps in the packet-number sequence they receive on the multicast flow, though it remains under the ACK cap of $C = 50$ Mbps.

Takeway. FCQUIC$_A$ performs similarly or better than NORM in the presence of losses, suggesting that its per-receiver unicast retransmission mechanism is not a bottleneck in its design, because it allows non-impacted receivers to continue receiving new application data.

VI. Related work

Transport protocols. We build on Flexicast QUIC [34], and focus on streaming use cases. We further extended FCQUIC with our adaptive acknowledgment design and file-rolling systems and evaluated the protocol for software updates. Other suggestions to extend QUIC with multicast have been proposed [44]–[46]. HTTP over multicast QUIC [44] was the first approach to deliver HTTP content over multicast using QUIC, but the design never led to a working implementation. MCQUIC [45], [46] suggested modifications in the QUIC protocol to support multicast. In this design, receivers maintain a unicast connection with fan-out servers, while opening another connection for content delivery via multicast. The specification focuses more on source authentication than on reliability and congestion control, which are mandatory for software updates. Moreover, we could not find a complete implementation of MCQUIC. FLUTE [26] is another protocol that disseminates large software updates over unidirectional transport, targeting multicast networks. It uses the concept of *carousel*, which is similar in spirit to our *file-rolling* approach. Similar to NORM [25], FLUTE does not have per-receiver unicast connections and does not use (N)ACK from receivers to detect losses. It uses Forward Error Correction (FEC) codes to improve reliability. Features such as data rate adaptation are left to the application, rendering FLUTE complex to deploy.

Acknowledgment rate adaptation. NORM minimizes the acknowledgment rate by using NACKs with an echo-suppression mechanism. Previous research included adding intermediate nodes to aggregate acknowledgments [27]. These approaches assume that the source has no knowledge of the multicast group's size— an assumption that led to the design of IP Multicast and related protocols.

Chunking files into smaller pieces has already been explored for Peer-to-Peer systems such as BitTorrent [47]. These systems allow receivers to download parts of the file concurrently, improving performance. Although the block size was not initially investigated, studies have shown its impact [48]. They highlighted that a smaller size leads to better performance for small downloads, but reducing it too much increases overhead. We observed the same conclusions in Section V-E.

VII. Discussion

Software updates and releases of popular games sporadically cause spikes in network utilization for Internet Service Providers (ISPs), requiring them to add more capacity. These peaks are caused by the exclusive usage of unicast distribution from Content Delivery Network edge servers to deliver these large files. In this paper, we reconsider the use of multicast protocols to efficiently distribute these updates to large groups. We investigate the ACK implosion problem and derive how we can *dynamically control* the acknowledgment rate by leveraging the additional knowledge gained from the unicast paths maintained in Flexicast QUIC. Through emulation using CloudLab [28], we demonstrate the flexibility of our approach across various scenarios, underlining its benefits over unicast QUIC and comparable multicast protocols such as NORM. Because the testbed limits the scale of our experiments, we analyze the CPU load of our FCQUIC server and derive a model to estimate this load for larger sets of receivers: we show that the major factor influencing the CPU load is indeed the acknowledgment rate, which we control in our extension of FCQUIC. We also discuss the trade-offs and limits of our approach, showing that hard limits on the acknowledgment rate can degrade communication quality.

As part of our future work, we will explore deployment questions and the requirements from ISPs to enable multicast forwarding within their networks for external entities such as CDN PoPs. We are currently discussing with GEANT, the European research network, to deploy our solution on our university campus network to distribute software updates via multicast. To this end, we will also extend our current approach using HTTP/3 to use the HTTP PUSH method [49], which provides cleaner semantics.

We release the source code of FCQUIC$_A$, the experiment script, details to reproduce the results, and the raw results used in this paper: https://github.com/louisna/fcquic-ack-experiments-ToN.

APPENDIX A
FILE-ROLLING BLOCKS TO QUIC STREAMS

Consecutive streams of the same type thus have identifiers incremented by 4. Since the file-rolling streams are unidirectional and initiated by the server, the two lowest bits are always set to `0b11`. Repeated blocks are sent in consecutive streams. As such, the source sends again the content of B_1 in stream $s_4 = s_3 + 4 = s_1 + 4 * n_B$, B_2 on $s_5 = s_3 + 4 + 4 = s_2 + 4 * n_B$,... Using numerous stream identifiers in QUIC is encouraged, since identifiers range up to $2^{62} - 1$. Following QUIC's semantics [5], the source can repeat the file $n = \frac{2^{62}-1}{4*n_B}$ times, where n_B is the number of blocks. For three blocks, $n > 3.84 * 10^{17}$.

REFERENCES

[1] Unigamesity, "How big is fortnite?" 2025. [Online]. Available: https://www.unigamesity.com/how-big-is-fortnite/

[2] O. G. AJ Curchill, "Discover playstation game install sizes: A reference guide," 2025. [Online]. Available: https://outsidergaming.com/playstation-game-install-sizes-reference-guide

[3] M. Claypool, A. Kica, A. La Manna, L. O'Donnell, and T. Paolillo, "On the impact of software patching on gameplay for the league of legends computer game," *The Computer Games Journal*, vol. 6, no. 1, pp. 33–61, 2017.

[4] W. Eddy, "Transmission Control Protocol (TCP)," RFC 9293, Aug. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9293

[5] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[6] F. L. (Namex), "Yesterday fortnite launched a massive update that significantly impacted the network, visible also through ixps," 2025. [Online]. Available: https://www.linkedin.com/feed/update/urn:li:activity:7258748607485386753/

[7] openreach, "Fortnite patches boost record-breaking uk broadband traffic," 2025. [Online]. Available: https://www.openreach.com/news/fortnite-patches-boost-record-breaking-uk-broadband-traffic/

[8] J. Blendin, F. Bendfeldt, I. Poese, B. Koldehofe, and O. Hohlfeld, "Dissecting apple's meta-cdn during an ios update," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 408–414.

[9] F. L. (Namex), "On the occasion of microsoft patch tuesday," 2025. [Online]. Available: https://www.linkedin.com/feed/update/urn:li:activity:7262486167533592577/

[10] ——, "Have you ever seen the shape of an apple ios update?" 2025. [Online]. Available: https://www.linkedin.com/feed/update/urn:li:activity:7242078765726502913/

[11] ——, "Namex steps into the terabit era," 2025. [Online]. Available: https://labs.ripe.net/author/flavio_luciani_1/namex-steps-into-the-terabit-era/

[12] R. van der Berg, "Is 8 terabit per second too much traffic?" 2025. [Online]. Available: https://www.linkedin.com/posts/rudolfvanderberg_is-8-terabit-per-second-too-much-traffic-activity-7302579675208273920-i9jX/

[13] Akamai, "Akamai global infrastructure," 2026. [Online]. Available: https://www.akamai.com/why-akamai/global-infrastructure

[14] C. of Duty Staff, "The road to black ops 6 launch: Preload and new ui," 2024. [Online]. Available: https://www.callofduty.com/blog/2024/10/call-of-duty-black-ops-6-road-to-launch-preload-new-ui-intel

[15] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the ip multicast service and architecture," *IEEE network*, vol. 14, no. 1, pp. 78–88, 2000.

[16] Microsoft, "Use multicast to deploy windows over the network with configuration manager," 2022. [Online]. Available: https://learn.microsoft.com/en-us/mem/configmgr/osd/deploy-use/use-multicast-to-deploy-windows-over-the-network

[17] M. Stoicescu, G. Aubert, F. Borgia, O. Espanyol, M. Higgins, M. Horny, D. Lee, P. Miu, I. Parodi, A. Patchett *et al.*, "Reducing time to results with eumetsat's new data services," in *EGU General Assembly Conference Abstracts*, 2020, p. 10267.

[18] McGill, "Using multicast for largescale live streaming over the internet," 2025. [Online]. Available: https://datatracker.ietf.org/meeting/124/materials/slides-124-mops-side-mtg-using-multicast-for-large-scale-live-streaming-over-the-internet-01

[19] S. A. (BT), "Multicast-assisted unicast delivery," 2025. [Online]. Available: https://datatracker.ietf.org/meeting/123/materials/slides-123-mops-multicast-assisted-unicast-delivery-maud-introduction-00.pdf

[20] J. Zhang, W. Cheng, and K. Liu, "Requirements and Gap Analysis of Multicast in AI Data Centers," Internet Engineering Task Force, Internet-Draft draft-zhang-rtgwg-multicast-requirements-gaps-aidc-01, Mar. 2026, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-zhang-rtgwg-multicast-requirements-gaps-aidc/01/

[21] B. Grönvall, A. Westerlund, and S. Pink, "The design of a multicast-based distributed file system," in *Symposium on Operating Systems Design and Implementation: 22/02/1999-25/02/1999*. USENIX-The Advanced Computing Systems Association, 1999.

[22] B. Grönvall, I. Marsh, and S. Pink, "A multicast-based distributed file system for the internet," in *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, 1996, pp. 95–102.

[23] S. Paul, K. K. Sabnani, J.-H. Lin, and S. Bhattacharyya, "Reliable multicast transport protocol (rmtp)," *IEEE journal on selected areas in communications*, vol. 15, no. 3, pp. 407–421, 1997.

[24] K. Obraczka, "Multicast transport protocols: a survey and taxonomy," *IEEE Communications magazine*, vol. 36, no. 1, pp. 94–102, 2002.

[25] J. P. Macker, C. Bormann, M. J. Handley, and B. Adamson, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol," RFC 5740, Nov. 2009. [Online]. Available: https://www.rfc-editor.org/info/rfc5740

[26] T. Paila, R. Walsh, M. Luby, V. Roca, and R. Lehtonen, "FLUTE - File Delivery over Unidirectional Transport," RFC 6726, Nov. 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6726

[27] B. N. Levine and J. Garcia-Luna-Aceves, "A comparison of known classes of reliable multicast protocols," in *Proceedings of 1996 International Conference on Network Protocols (ICNP-96)*. IEEE, 1996, pp. 112–121.

[28] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb *et al.*, "The design and operation of {CloudLab}," in *2019 USENIX annual technical conference (USENIX ATC 19)*, 2019, pp. 1–14.

[29] R. Singh, A. Dunna, and P. Gill, "Characterizing the deployment and performance of multi-cdns," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 168–174.

[30] J. Maisonneuve, M. Deschanel, J. Heiles, W. Li, H. Liu, R. Sharpe, and Y. Wu, "An overview of iptv standards development," *IEEE Transactions on broadcasting*, vol. 55, no. 2, pp. 315–328, 2009.

[31] M. Mellia and M. Meo, "Measurement of iptv traffic from an operative network," *European Transactions on Telecommunications*, vol. 21, no. 4, pp. 324–336, 2010.

[32] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[33] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind, "Managing multiple paths for a QUIC connection," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-multipath-21, Mar. 2026, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/21/

[34] L. Navarre, Q. De Coninck, T. Barbette, and O. Bonaventure, "Taking the best of multicast and unicast with flexicast quic," *ACM SIGCOMM Computer Communication Review*, vol. 55, no. 2, pp. 2–12, 2025.

[35] S. Pingali, D. Towsley, and J. F. Kurose, "A comparison of sender-initiated and receiver-initiated reliable multicast protocols," *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, no. 1, pp. 221–230, 1994.

[36] R. T. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122, Oct. 1989. [Online]. Available: https://www.rfc-editor.org/info/rfc1122

[37] J. Iyengar, I. Swett, and M. Kühlewind, "QUIC Acknowledgment Frequency," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-ack-frequency-14, Feb. 2026, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency/14/

[38] Cloudflare, "Quiche: savoury implementation of the quic transport protocol and http/3." 2025, https://github.com/cloudflare/quiche.

[39] IPNetworkingLab, "Flexicast quic implementation, based on cloudflare quiche," 2025. [Online]. Available: https://github.com/IPNetworkingLab/flexicast-quic

[40] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind, "Multipath Extension for QUIC," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-multipath-11, Oct. 2024, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/11/

[41] L. Foundation, “Data plane development kit (DPDK),” 2015. [Online]. Available: http://www.dpdk.org
[42] U. N. R. Laboratory, “Nack-oriented reliable multicast (norm) implementation and tools (rfcs 5740, 5401),” 2025, https://github.com/USNavalResearchLaboratory/norm.
[43] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle, “Quic on the highway: Evaluating performance on high-rate links,” in *2023 IFIP Networking Conference (IFIP Networking)*. IEEE, 2023, pp. 1–9.
[44] L. Pardue, R. Bradbury, and S. Hurst, “Hypertext Transfer Protocol (HTTP) over multicast QUIC,” Internet Engineering Task Force, Internet-Draft draft-pardue-quic-http-mcast-11, Jul. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-pardue-quic-http-mcast/11/
[45] M. Franke, J. Holland, and S. Schmid, “Mcquic-a multicast extension for quic,” in *2024 22nd International Symposium on Network Computing and Applications (NCA)*. IEEE, 2024, pp. 185–192.
[46] J. Holland, L. Pardue, and M. Franke, “Multicast Extension for QUIC,” Internet Engineering Task Force, Internet-Draft draft-jholland-quic-multicast-07, Jul. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-jholland-quic-multicast/07/
[47] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, “The bittorrent p2p file-sharing system: Measurements and analysis,” in *International workshop on peer-to-peer systems*. Springer, 2005, pp. 205–216.
[48] P. Marciniak, N. Liogkas, A. Legout, and E. Kohler, “Small is not always beautiful,” *arXiv preprint arXiv:0802.1015*, 2008.
[49] M. Thomson and C. Benfield, “HTTP/2,” RFC 9113, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9113