Improving the Efficiency and Flexibility of Network Communications

Louis Navarre

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

August 2026

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Charles **Pecheur** (Chair)	UCLouvain/ICTEAM, Belgium
Pr. Etienne **Rivière**	UCLouvain/ICTEAM, Belgium
Pr. Ramin **Sadre** (Secretary)	UCLouvain/ICTEAM, Belgium
Pr. Jörg **Ott**	TUM, Germany
Pr. Matthias **Wählisch**	TU Dresden, Germany
Pr. Olivier **Bonaventure** (Advisor)	UCLouvain/ICTEAM, Belgium

Improving the Efficiency and Flexibility of Network Communications

by Louis Navarre

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was partially supported by the F.R.S-FNRS.

A clever person solves a problem.
A wise person avoids it.
ALBERT EINSTEIN

Use of Artificial Intelligence

No large language models (LLMs) have been used to generate the text of this thesis. We used tools such as Grammarly to correct grammatical mistakes and improve the text's form based on the first human-written text. Anthropic's Claude Opus 4.8 has been used to generate some Figures from the background, namely Figures 1.7, 1.8, 1.15 and 1.16.

The source code, evaluation scripts, and experimental results discussed in this thesis have not been generated or derived from such LLMs and constitute original work from the author.

Acknowledgments

First, I thank the jury members, Jörg Ott, Matthias Wählisch, Etienne Rivière, Ramin Sadre, and Charles Pecheur, for their time and valuable comments that improved the content of this thesis.

I want to thank my advisor, Olivier Bonaventure, for the opportunity to conduct research in his lab in such an inspiring environment. This thesis is the result of his crazy ideas (working on multicast starting in 2021 is one of them), his unfailing support, and the research freedom he gave me. All those small details, such as ice cream breaks during the summer and debriefs of cycling races (especially the "alors Remco?" during the Vuelta 2022), created an enjoyable working environment and will stay as good memories. Olivier's impact goes far beyond research, as he is an example of kindness to others, making UCLouvain a better place for researchers and students.

Other people also provided valuable help during this thesis. Prof. Matthias Wählisch, thank you for constantly challenging me since December 2022, and for hosting me at TU Dresden in 2025. Prof. Tom Barbette, thank you for your involvement in my research since my first day at UCLouvain. Prof. Quentin De Coninck, I truly believe that my approach to research shifted after our meeting in September 2024, and your contributions helped my thesis take shape and move forward. Lucas Pardue, thank you for the opportunities to present my work to the QUIC WG during the IETF meetings, especially given the tight schedules. I also want to thank the INGI staff, Vanessa, Anthony, Nicolas, and Jean-Marc, for all your help making my stay in the department as smooth as possible despite all the requests I made for my research.

To François, Maxime, Nicolas, and Thomas: you made these years unforgettable. I thank you for the "3 glorieuses", as we say, formed of insightful discussions, *rigopleurages*, coffee breaks, and intense conferences. Thank you for all the knowledge, advice, and help you have given me. Your presence was one of the main reasons I wanted to come to the office all these years. I don't forget the other members of the office with whom I share good memories: Vany, Tom, Matthieu, Anthony, Paul-Edouard. To Vany especially, you exemplify what a top colleague is all about. Thank you for the quality time we have spent together and for understanding and supporting my humor. Similar to what my former colleagues gave me, I hope I have shared all my experience with you. I am already amazed by what you achieved in so little time, and I'm sure that you will succeed in whatever you aim for in the future.

To François, I want to thank you for all you did outside of my research. You are a source of inspiration as a human being, and I cherish every opportunity to learn from you. Our discussions, your experience, and your advice made me a better person.

I also want to thank everyone at INGI who contributed to a good atmosphere at work and during outside activities: Olivier G and Mathias for our numerous discussions and our sport sessions; François DK for introducing me to the music universe, for the festivals and concerts, and just for being a human full of good energy; Nikita for being the sunshine of the department; Donatien for your anecdotes and good vibes; Augustin and Alexandre for the ragots; Gorby, for our sport discussions; Benoit for being the best Brusseleir; Clément for the coffee discussions; Maxime V for your beautiful t-shirts; Benoit R for introducing me to Hades; Lucile, Damien and Nicolas G for the trip to Vienna; Augustin C for the philosophical discussions; Achille for motivating me to learn the handstand; Hélène for the cheese and your amazing culinary skills; Yinan for the Chartreuse; Rémi for your involvement in the life of the department; but also all the others, Alice, Aurélien, Juliette, Edouard, Julien, Charles, ...

Next, I wish to thank my friends, Gilles, Thibault, Nicolas, Edgar, Nahi and Guilhem, who accompanied me and helped our friendship evolve despite my lack of initiative over the past few years. You have been there during the toughest events of my life, and you have contributed to so many memories that coincide with this thesis. Thank you to all my housemates, with whom I spent so much quality time over the last two years; Macha and Maïté for the discussions in the kitchen; Midou, Hadri and Romain for life at the *coloc*.

To my family, thank you for your support. Charles, thank you for being my example for as long as I can remember and for being an inexhaustible source of jokes; Mom, I remember you were the first one to push me to pursue a PhD. At first, I just wanted to do the opposite, because that's what children do. Five years later, I must admit you were right. Thank you.

Finally, I thank Emma, for whom the last few years have been full of joy and laughter. You once told me that we must cherish today, as we don't know what tomorrow will bring. This made me realize that life is not all about working, which, paradoxically, made me more efficient at work, but also made life more enjoyable. Thank you for supporting my late-night working sessions and my varying moods, for understanding me when I'm not talking, and, foremost, for being so kind to the people around you.

This thesis would not have been possible without all of you. Through a joke, a gesture, an insightful discussion, or quality time, you helped me during these last five years. "Life is about making memories" (from François), and these years are full of good memories. For this, I warmly thank you all.

Contents

Preamble

The Internet is one of the most prevalent tools in today's society. What started as a research project now connects around 6 billion users worldwide, almost three-quarters of the world's population [ITU2026]. The Internet is a shared architecture that interconnects more than 75 k networks [CIDR2026; Hus25]. As the Internet grew, use-cases emerged with ever-stronger bandwidth and latency requirements. While initial applications were emails, short file transfers and remote terminal accesses [Rob88], the improvement in quality of service offered by today's networks enables much more powerful use-cases, such as immersive live streaming through virtual reality.

One such range of applications involves sharing the same content with numerous clients simultaneously, known as *one-to-many* communications. These scenarios typically involve live-streaming events, e.g., sports matches, since all clients want to receive the stream as quickly as possible, but they also occur during the release of popular software, such as system updates [Ble+18] and video games [ope25]. While these large one-to-many communications occur sporadically, studies show they are becoming more frequent [Li+22]. The combination of ever-increasing quality of service and large one-to-many distribution induces a paradox, which is the starting point of this thesis:

> To ingest sporadic, large one-to-many communications, network providers must over-provision their networks far beyond daily requirements. As such, networks are most of the time underutilized and struggle to offer a consistent quality of service for these one-to-many applications.

Examples showing this paradox are not isolated and represent different entities. For example, Akamai, one of the largest Content Distribution Networks (CDN), reached multiple times more than 250 Tbps of traffic at peak times [Aka22], and reported that between 20 % and 50 % of it is *duplicate* traffic [Hol20]. Vodafone, a large telecommunications company, reported that its network scales to ingest up to 8 Tbps of traffic, while daily requirements are only 4 Tbps, and that this over-provisioning is especially due to live streaming [Ber25]. During this thesis, we studied the evolution of the network of OVH, a large European Cloud Provider, which exposes its network through a *weathermap* [OVH]. We reported a similar over-provisioning, i.e., the network uses less than 50 % of its capacity more than 99 % of the time [Pir+22b].

This thesis starts from the aforementioned observation and suggests methods to improve the efficiency of communication. In the first part, we leverage the network's over-provisioning to propose a network-layer tunnel supporting Forward Erasure Correction (FEC), a form of channel coding, that reduces application latency transparently. Instead of relying on transport-layer retransmissions, which require time to detect lost data, FEC injects redundant packets into the network *a priori* to recover from losses more quickly.

In the second part, we focus on the principal factor that necessitates network over-provisioning to ingest large-scale one-to-many communications: how servers distribute content to their clients, using *unicast* transport protocols. Another possibility is to rely on *multicast* protocols, which guarantee efficient data distribution by replicating data where necessary inside the network to reach all intended receivers. Despite the benefits of the approach, multicast suffers from severe deployment issues [Dio+00], which hinder its widespread adoption on the Internet. In this thesis, we reconsider the use of multicast for large-scale one-to-many communications. We focus on the transport layer by starting from QUIC [RFC9000], a modern protocol that reflects the knowledge gained from several decades of experience with TCP. We extend QUIC with *flexible multicast*, a paradigm that benefits from multicast when available and from unicast otherwise. We explore multicast transport-layer questions such as reliability, congestion control, and server scalability.

Multicast communication was proposed almost 40 years ago and has been extensively studied since then. Several multicast transport protocols have been proposed, each aiming to provide scalable delivery assuming the support from a multicast-capable underlying network. The failure to deploy multicast on the Internet – also due to routing problems that we mention but do not explore in this thesis – thus led to the *all-or-nothing* problem, where multicast works if and only if the source and all receivers have end-to-end multicast connectivity. Since then, researchers have proposed overlay networks by implementing multicast at the application layer on top of unicast routing [CRZ00], with limited success. What has been less studied is the possibility of combining multicast and unicast delivery within the same reliable transport protocol, allowing a fallback to unicast delivery if multicast is unavailable. Clients that benefit from end-to-end multicast connectivity with the server could leverage it to improve the underlying network's resource usage. The others can rely on unicast to ensure data delivery directly from the source (i.e., without requiring support from other entities). Closely integrating unicast and multicast within a single transport raises several questions, including those about reliability and congestion control. This thesis pushes these interactions further, shifting the paradigm of multicast transport: clients should seamlessly transition between robust unicast and efficient multicast, e.g., based on their network quality. Thinking about multi-

cast transport this way also enables us to reconsider other historical issues, such as how to guarantee a minimal quality of service for multicast members in the presence of a single or a subset of bottleneck receivers. This design is further made possible with the development of modern protocols such as QUIC [RFC9000] and its Multipath extension [Liu+26].

An interesting question is whether supporting multicast should be implemented at the transport or application layer. Indeed, applications can implement all features exposed by the transport layer, such as reliability, congestion control, and flow control. Similarly, the application layer can be responsible for using multicast when available and unicast otherwise, at the cost of more implementation complexity. QUIC [RFC9000] demonstrates that these mechanisms, implemented in the transport, offer easier interaction for the application. As we will analyze later in this thesis, *flexible multicast* heavily relies on features exposed by Multipath [Liu+26], which is implemented in the transport layer [WHB08]. With flexible multicast implemented in the transport, applications can rely on a single protocol that simultaneously guarantees efficiency (multicast) and fallback (unicast).

While this thesis focuses solely on multicast at the transport layer, we acknowledge that the widespread adoption of the mechanisms discussed in some of these chapters is closely linked to multicast issues at different layers, which involve routing and economic issues [Dio+00]. Recent advancements in multicast routing, through Bit Index Explicit Replication (BIER) [RFC8279] and Automatic Multicast Tunneling (AMT) [RFC7450], aim to provide such solutions. It is interesting to note that AMT, similarly to flexicast, shifts the paradigm from the *all-or-nothing* problem to incremental multicast support, by leveraging efficient multicast when available, and using unicast tunneling for receivers with unicast-only support.

This thesis is split into seven chapters.

- Chapter 1 introduces the concepts needed to understand the thesis. It covers transport-layer mechanisms introduced with TCP, and presents the QUIC protocol and its soon-to-be-standardized multiple paths management extension. The chapter also presents Forward Erasure Correction (FEC) to recover from packet losses. Finally, we detail how multicast protocols enable efficient data delivery, the challenges arising from this communication method that hindered its deployment, and recent advancements enabling us to reconsider its use today.

- Chapter 2 suggests utilizing the spare bandwidth stemming from network over-provisioning by exposing a tunnel transparent for applications – and their underlying transport – aiming to reduce latency in the presence of packet losses. We present the High-speed Robust Tunnel (HIRT), a tunnel powered by IPv6 Segment Routing to dynam-

ically inject redundant information at the tunnel ingress through Forward Erasure Correction techniques, allowing for faster recovery of packet losses than relying on transport-level retransmissions. We design an adaptive FEC scheduler that adjusts redundancy based on network feedback from the tunnel egress. To avoid injecting redundant data when spare bandwidth is unavailable, HIRT detects router overload and prioritizes packet forwarding over generating redundant data that may further contribute to node overload. We implement HIRT in software and show that our optimizations allow us to protect >50 Gbps of traffic under heavy losses. We further evaluate HIRT over Starlink, a Low Earth Orbit network that suffers from non-congestion-induced losses [Mic+22b], and highlight the benefits for latency-sensitive applications.

- Chapter 3 reconsiders multicast transport protocols by proposing *flexible multicast*, i.e., *flexicast*, as a close interplay between efficient multicast and robust unicast within a single protocol, with seamless transitions between the two. We introduce Flexicast QUIC (FCQUIC), an extension of QUIC [RFC9000] leveraging its multiple path management mechanism (Multipath QUIC in short) [Liu+26]. The design includes reliability and congestion-control mechanisms and provides a path scheduler that seamlessly detects multicast link failures and bottleneck receivers, falls back to unicast delivery, and reintegrates the multicast group when possible. We provide an initial implementation of FCQUIC based on the Multipath QUIC specifications from October 2024 [Liu+24], and show the robustness enabled by the flexible multicast paradigm. Next to the clear benefits of multicast over unicast in network usage, we present benchmarks showing that FCQUIC improves server scalability compared to QUIC alone.

- Chapter 4 builds upon Flexicast QUIC to investigate the *ACK implosion* problem of multicast, where a single data packet triggers acknowledgments from all receivers, potentially overwhelming the source. We design an adaptive delayed strategy to cap the acknowledgment rate from receivers based on the data rate and the group size. Because our testbed limits the scale of our experiments, we model the CPU load of an FCQUIC source, enabling theoretical extrapolation of benchmarks to larger experimental setups. We also provide a more efficient implementation of Flexicast QUIC, based on the experience from Chapter 3 and with Multipath QUIC starting from July 2025 [Liu+26].

- Chapter 5 constitutes an additional contribution that arose from the discussions initiating the work of Chapter 4. A common strategy to limit

ACK implosion is to rely on negative acknowledgments (NACKs), i.e., reporting only missing ranges of packet numbers. However, the QUIC specification states that emitters can *skip packet numbers* when sending packets [RFC9000], which would instead induce NACK implosion. The objective behind this *packet skipping* feature is to mitigate the *optimistic acknowledgment* attack, in which a client fakes acknowledgments to the server, forcing it to increase its sending rate and thus saturating the network. In this chapter, we analyze the optimistic acknowledgment attack in QUIC, showing that despite being part of the specification, the *packet number skip* was not widely implemented, and that numerous QUIC implementations were vulnerable, including production-ready stacks from tech giants deployed in real servers. We show how to mount this attack to force all vulnerable stacks to effectively increase their sending rate to a level significantly higher than what the underlying network can support, and show how the *packet number skip* feature mitigates it.

- Chapter 6 takes a step back from the contributions of this thesis, highlighting future research questions that emerged from this work.
- Chapter 7 concludes this thesis.

Bibliographic notes

This thesis led to the following publications:

Conference Publications

1. L. Navarre, F. Michel, and T. Barbette. "A High-Speed Robust Tunnel Using Forward Erasure Correction in Segment Routing". In: *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*. IEEE. 2024, pp. 1–12.

Journal Publications

1. L. Navarre, Q. De Coninck, T. Barbette, and O. Bonaventure. "Taking the best of multicast and unicast with flexicast QUIC". in: *ACM SIGCOMM Computer Communication Review* 55.2 (2025), pp. 2–12.

Workshop Publications

1. L. Navarre and O. Bonaventure. "MAY is not enough! QUIC servers SHOULD skip packet numbers". In: *Proceedings of the 2025 Applied Networking Research Workshop*. 2025, pp. 136–142.

Posters and Demos

1. L. Navarre and O. Bonaventure. "Towards an Internet Deployment of Flexible Multicast QUIC". in: *Proceedings of the ACM SIGCOMM 2025 Posters and Demos*. 2025, pp. 137–139.

    - Received the second place at the ACM SIGCOMM 2025 Student Research Competition.

Under submission

1. L. Navarre and O. Bonaventure. "Scaling Flexicast QUIC for Large-Scale Software Distribution Through Adaptive Acknowledgments". In: (2026).

Artefacts

1. The code of HIRT (Chapter 2) and the scripts for the experiments are publicly available [NMB24].

2. The code of Flexicast QUIC (Chapter 3), which extends Cloudflare *quiche* and the multipath branch [QUICHE; QUICHE-MULTIPATH] is publicly available [Nav+25]. We also release all the experiment scripts used to evaluate Flexicast QUIC in the same repository.

3. The code of the new version of Flexicast QUIC is based on a more recent version of Multipath [Liu+26] and with the adaptive acknowledgment delay [Nav26b], as well as the scripts used for the experiments and the raw results [Nav26a].

4. The scripts used during the ACM SIGCOMM 2025 demo session [NB25c; Nav25].

5. The evaluation scripts and raw results of our analysis of the Optimistic Acknowledgment (OACK) attack in QUIC (Chapter 5), as well as our modified Cloudflare *quiche* client performing the OACK attack, and the proposed patch for the same implementation [NB25b].

Miscellaneous Contributions

1. L. Navarre, F. Michel, and O. Bonaventure. "SRv6-FEC: bringing forward erasure correction to IPv6 segment routing". In: *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 2021, pp. 45–47.

2. L. Navarre, F. Michel, and O. Bonaventure. "It Is Time to Reconsider Multicast". In: *IAB workshop on Environmental Impact of Internet Applications and Systems*. Vol. 2022. 2022.

3. M. Jadin, Q. De Coninck, L. Navarre, M. Schapira, and O. Bonaventure. "Leveraging eBPF to make TCP path-aware". In: *IEEE Transactions on Network and Service Management* 19.3 (2022), pp. 2827–2838.

4. M. Piraux, T. Barbette, N. Rybowski, L. Navarre, T. Alfroy, C. Pelsser, F. Michel, and O. Bonaventure. "The multiple roles that IPv6 addresses can play in today's internet". In: *ACM SIGCOMM Computer Communication Review* 52.3 (2022), pp. 10–18.

5. M. Piraux, L. Navarre, N. Rybowski, O. Bonaventure, and B. Donnet. "Revealing the evolution of a cloud provider through its network weather map". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 298–304.

6. Q. De Coninck, L. Navarre, and N. Rybowski. "On integrating ebpf into pluginized protocols". In: *ACM SIGCOMM Computer Communication Review* 53.3 (2024), pp. 2–8.

Background 1

This chapter introduces all the concepts that we use throughout this thesis. It also discusses the state-of-the-art techniques that we build upon in the subsequent chapters.

1.1 Connecting two hosts over a network

Hosts use *interfaces* to communicate with each other, similarly to humans having multiple ways to communicate with each other, e.g., their mouths and hands. These interfaces rely on different underlying technologies, such as RJ45 cables, Wi-Fi, and Bluetooth. For example, two computers can directly transmit data if they are connected through their Bluetooth adapter. The spread of communication technologies did not stop at having two devices communicate directly. As such, these electronic tools often rely on a *network* to carry their messages from one host to another, even if they are not directly connected. Figure 1.1 illustrates a simplified view of two end-hosts indirectly connected through a network. A network is essentially composed of *routers*, which forward packets from a source to a destination. In that regard, we can view the Internet as a gigantic entity composed of numerous networks. Once a device is connected to the Internet, e.g., through its provider, it can contact any other end-host also connected to the Internet. There are multiple points worth discussing in Figure 1.1, from a high-level perspective.

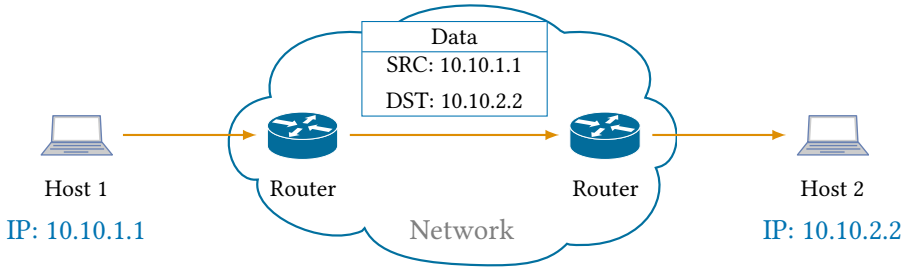


Figure 1.1: Two hosts communicate through a network.

1.1.1 IP addresses identify devices on a network

The interfaces of end-host devices and routers are uniquely identified in a network. Computer networks rely on the Internet Protocol, in which the interfaces of devices are identified by an *IP address*. The first deployment of the Internet relied on IPv4 [RFC791], where an IP address spans 32 bits, giving roughly 4.3 billion possible addresses. Some of these ranges were reserved for specific purposes – such as *multicast*, which we detail later in this chapter. At the time IPv4 was designed, the Internet was only a research project connecting at most a few hundred hosts, and that limited address space was not a concern. However, this limitation became apparent with the exponential growth of the Internet, and the IPv4 address space rapidly became exhausted, prompting the development of IPv6 [RFC8200], which uses 128-bit addresses. Devices exchange *packets* to communicate, which contain, among other things, the source and destination IP addresses, allowing routers to forward them. The *Maximum Transmission Unit* (MTU) defines the maximum packet size that an interface supports. In Ethernet networks, the MTU is usually 1500 B. IP addresses are allocated hierarchically, starting from the IANA, which maintains the global pool of IP addresses.

1.1.2 Network protocols

All these devices must agree on *how to communicate* with each other. If we take one more look at our previous analogy, humans developed language to transmit information to one another. For electronic devices, this agreement takes the form of *network protocols*, i.e., sets of defined rules governing the exchange of data. In the domain of global, interoperable, and open communication, these protocols usually take the form of *specifications* written in "Request For Comments" (RFC) documents, which use a precise notation to describe each protocol, namely its purpose, scope, and behavior. As an example, the document describing IPv6 is RFC 8200 [RFC8200], which defines, among other things, its header and address length, its placement within network packets, and the operations one can perform on such packets.

The Internet Engineering Task Force is the organization responsible for defining and writing these RFC documents. Each RFC describing a network protocol allows engineers to implement their own stack, e.g., in a specific framework or programming language, while ensuring interoperability.

1.1.3 Routing protocols

Routing protocols maintain the tables used to forward packets from a source to a destination. Returning to Figure 1.1, once the two end-hosts have their IP addresses, they do not know *how* to reach each other. They only know their

peer's address. Multiple strategies exist to send packets to the destination, e.g., broadcasting the data to all nodes in the network, which results in inefficient resource utilization. Static routing is another strategy where the routes are statically defined and installed on routers. This strategy works well but suffers from network topology changes and failures, as nodes cannot dynamically learn new paths to reach the destination. A final strategy is to rely on *routing protocols* to learn the path towards other nodes in the network.

Intra-domain protocols run inside a specific domain, such as an Internet Service Provider network. By exchanging specific information, the routers in the domain learn how to reach other nodes inside the domain. Routers forward packets along the shortest path toward the destination based on the IP destination address. OSPF [RFC5340] and IS-IS [RFC1195] are two examples of intra-domain routing protocols.

Inter-domain protocols. The Internet is a large coordination of numerous intra-domain networks. While each domain runs its own services with its own configuration, networks must coordinate to ensure global delivery of data across the planet. BGP [RFC4271] is the inter-domain protocol used on the Internet.

1.2 Transport protocols

IP networks provide a *best-effort* service to end-hosts, meaning that routers forward packets as efficiently as possible based on current resource availability, but *without guarantee* of packet reliability, latency, or order, as packets may be altered during transmission (e.g., due to transient congestion). Depending on the use-case, hosts may require such ordered, reliable data delivery. For this, they rely on *transport protocols*.

Operating systems expose several transport protocols to end-hosts with different service guarantees. The interface between the kernel and the application is commonly known as the Socket API [SOCKET]. This API abstracts the transport-layer mechanisms to the application, providing the following contracts, depending on the Socket type, which can either be SOCK_DGRAM or SOCK_STREAM:

- SOCK_DGRAM: support for unreliable, unordered *datagrams*. A datagram is a message of a fixed length, which must fit within the MTU of the underlying network.

- SOCK_STREAM: support for a reliable, ordered *stream*. This type establishes a connection between the two hosts and opens a bidirectional byte stream. The SOCK_STREAM contract ensures that our peer will reliably receive the data, in order, within a reasonable amount of time, or the connection will close, indicating that the transmission failed. In

contrast to datagrams, there are no strict boundaries, as the sent data conceptually flows in a bidirectional single stream of bytes from the source to the destination, even though the protocol chunks them into individual packets to fit in the MTU of the network.

Behind these two abstractions, the operating systems use the UDP [RFC768] protocol for SOCK_DGRAM and TCP [RFC9293] for SOCK_STREAM.

Application multiplexing and the use of ports. In addition to ensuring end-to-end data delivery, transport protocols also enable application *multiplexing* through the use of *ports*: two end-hosts can communicate through different transport instances for distinct applications simultaneously. Ports are 16-bit unsigned integers. Some of them are reserved for specific protocols, for example, clients sending an HTTP [RFC2616] request to fetch an HTML page contact the server on port 80; the Secure Shell (SSH) [RFC4253] uses the port 22.

Chaining protocols. End-hosts use transport protocols to ensure end-to-end delivery, and rely on routers to forward them in a hop-by-hop manner, as described earlier. While the application data is embedded within the transport protocol, this transport-specific payload is encapsulated in an IP packet to let intermediate routers forward it correctly. This mechanism defines *protocol encapsulation*: routers process the IP packet, considering the inner payload as data, which potentially contains a TCP/UDP header.

1.3 The User Datagram Protocol

The User Datagram Protocol (UDP) [RFC768] is a simple transport protocol. Applications send UDP datagrams in a best-effort fashion. The receiver can process these datagrams as soon as it receives them, which may happen out of order or with losses. A datagram is a self-contained piece of information. UDP ensures that a datagram is either delivered entirely (i.e., no truncating), or not delivered at all, but does not guarantee reliability.

Figure 1.2 illustrates the UDP header format. The *Source* and *Destination* ports identify the source and destination ports of a UDP datagram, respectively. The *Length* field represents the length, in bytes, of the UDP datagram, including the header length. Finally, the *Checksum* contains the checksum computed over a pseudo-IP header comprising subfields of the IP outer header (including the IP source and destination addresses), the UDP header, with its checksum set to 0, and the UDP payload. The role of this checksum is to ensure that the data has not been corrupted during transmission. UDP is a *connectionless* protocol, meaning that no endpoint keeps transport state for the ongoing communication. The UDP header is 8 bytes long.

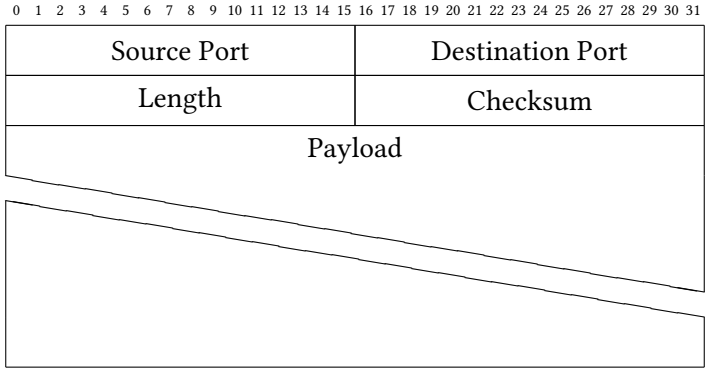


Figure 1.2: UDP header format. The header is fixed at 8 bytes.

1.4 The Transmission Control Protocol

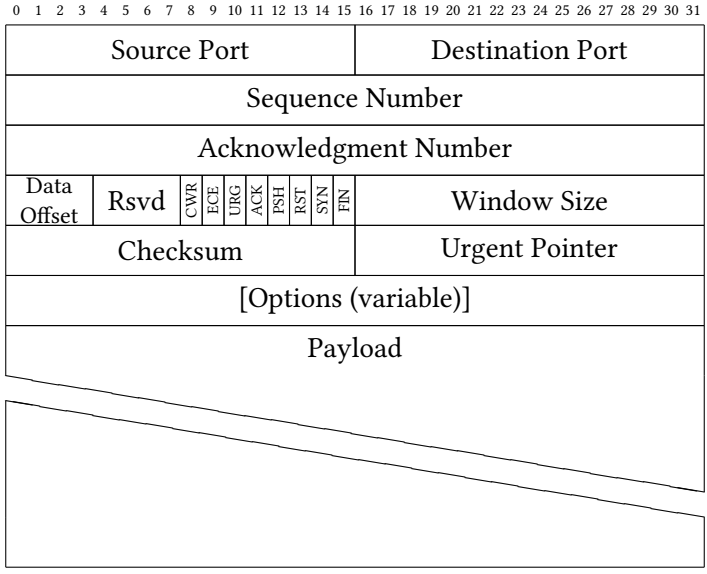


Figure 1.3: TCP header format. The minimum header size is 20 bytes (Data Offset = 5); options extend it up to 60 bytes.

The Transmission Control Protocol (TCP) is a connection-oriented, reliable transport protocol [RFC9293]. Figure 1.3 presents the TCP header format. To remain concise, we present only the most pertinent fields of the header for this thesis. The *Source* and *Destination Port* have similar meanings to those of UDP. The *Sequence Number* is the *shifted index* of the first byte of data of the byte stream contained in this packet, modulo 2^{32}. The *shifted index* is the

index plus the *initial sequence number*, a random value chosen by the endpoint during connection establishment. As we will see in the next section, the *Sequence Number* is used to ensure reliability. Conversely, the *Acknowledgment Number* is the shifted index of the next payload data byte we expect to receive from the peer's byte stream. The *Window* indicates the number of data octets that the endpoint is willing to accept, starting from the *Acknowledgment Number*. The *Checksum* has a similar meaning to UDP's checksum. TCP supports multiple options to extend the protocol's behavior. As such, the *Data Offset* field indicates the number of 32-bit additional words before the data payload. Finally, control bits in the TCP header add information used through the connection. As an example, the *FIN* bit is set whenever the connection can be safely closed; the *ACK* bit is set when the *Acknowledgment Number* value is significant; the *SYN* bit is used during connection establishment. We now present how TCP ensures reliable, ordered data transfer from connection establishment.

1.4.1 The three-way handshake

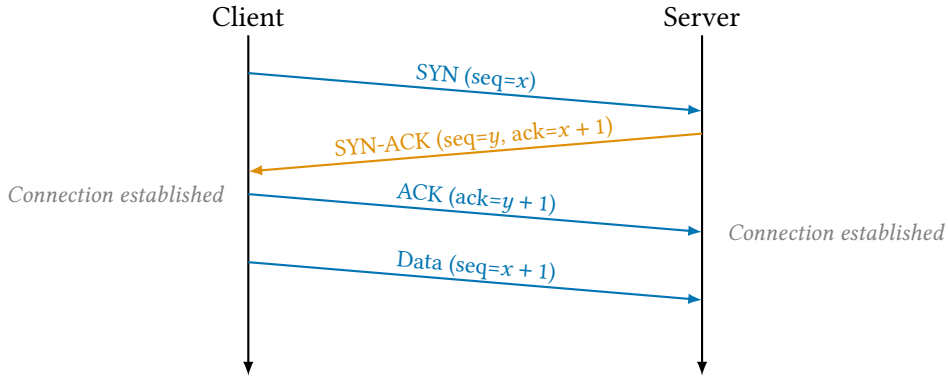


Figure 1.4: TCP's three-way handshake.

The *three-way* handshake, illustrated in Figure 1.4, is the method TCP uses to establish a connection. Connection establishment through a handshake has multiple purposes. First, it allows endpoints to verify that their peer uses the correct address, i.e., that the peer does not spoof another entity's IP address. Second, it allows them to negotiate the use of extensions (e.g., the Window Scaling extension [RFC7323]). For simplicity, we assume that the *client* initiates a connection to the server. The client starts by sending an empty TCP segment with the *SYN* flag set. This segment contains a *Sequence number* that the client chooses. If the server accepts the connection, it sends a TCP

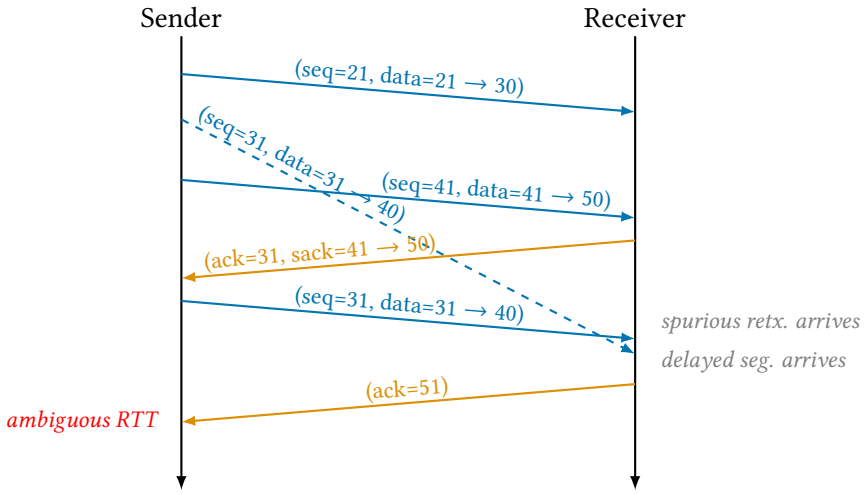


Figure 1.5: Spurious retransmissions due to packet reordering in TCP.

segment with the *SYN* and *ACK* flags, with the client's *Sequence Number* (increased by 1) echoed in the *Acknowledgment Number* field of the sent packet. In its turn, the server chooses its initial *Sequence Number*. The initial sequence number sent in the *SYN* packets by both hosts must be chosen to avoid session hijacking and prevent corruption from old, delayed packets. This three-way procedure allows hosts to detect duplicate connection initiations. For the client, the connection is established once it receives the *SYN+ACK* from the server; for the server, it is established upon reception of the *ACK* segment. No endpoint may send application data until it has established the connection. As such, the client must wait an entire round-trip before sending data (e.g., its HTTP request).

1.4.2 Reliable transfer

TCP endpoints rely on acknowledgments to guarantee correct data delivery. The *Acknowledgment Number* of the TCP header indicates the index of the next byte the peer expects to receive in the byte stream. Emitters use these acknowledgments and a *loss detection mechanism* to determine whether packets have been lost and need to be retransmitted.

The selective acknowledgments (SACKs) [RFC2018] option allows a TCP endpoint to indicate ranges of sequence numbers it received *out-of-order*, as illustrated in Figure 1.5. The next expected sequence number in the byte stream is 30, but the client already received `40..50`. Endpoints can advertise a maximum of 3 to 4 ranges of out-of-order received data, depending on the presence of other options [RFC2018].

With SACKs, it became possible to identify *gaps* in the acknowledgments. This ability to identify and retransmit only missing data is an instance of *Selective Repeat ARQ* (SR-ARQ), a reliability principle in which only lost or corrupted frames are retransmitted. A naive strategy would be to retransmit the data associated with the missed sequence number immediately. This strategy does not account for packet reordering, which can happen in IP networks. As illustrated in Figure 1.5, a delayed TCP segment may trigger SACKs, in turn triggering the retransmission of a TCP segment for data `30..40`, which was just delayed by the network. Such an event is called *spurious retransmission*, and contributes to inefficiencies: (i) these spurious retransmissions contribute to the network usage of the emitter, ultimately wasting bandwidth; (ii) the server cannot identify whether the initial TCP segment or the retransmission triggered the acknowledgment from the client, which makes its round-trip–time (RTT) estimation ambiguous. Over the years, several improvements have been proposed to TCP's retransmission strategy:

- The Retransmission Timeout (RTO) [RFC1122] is the amount of time the emitter waits without receiving any feedback before considering the sent segments lost. The emitter retransmits all unacknowledged data sent earlier than the expired RTO. The initial RTO is set to 1-3 seconds [RFC6298], but the value is adjusted to the real round-trip-time during the communication, using the formula [RFC6298]:

$$RTO = RTT_{smoothed} + K \times RTT_{var} \tag{1.1}$$

  where $RTT_{smoothed}$ and RTT_{var} respectively represent the smoothed RTT and the RTT variance. The Linux kernel uses $K = 4$.

- Fast Retransmit [RFC5681] is the first optimization that aims to detect losses rather than waiting for RTO expiration. Upon reception of 3 acknowledgments for the same expected next byte (i.e., three duplicate *Acknowledgment Number* values) while subsequent data is sent but not acknowledged, the TCP emitter retransmits in-flight data. Such duplicate acknowledgments are generated by the receiver when it processes out-of-order segments.

- Recent Acknowledgment (RACK) and Tail-Loss Probes (TLP) [RFC8985] are two recent optimizations improving TCP's loss detection and resilience to spurious retransmissions, aiming to replace Fast Retransmits for applications with limited in-flight data. Instead of counting (S)ACKs to infer losses, RACK uses per-segment timestamps and leverages SACKs to conduct time-based loss inferences [RFC8985]. The rationale behind TLP is to trigger ACK packets from the peer when the tail of packets in flight is lost, rather than waiting for the RTO to expire.

1.4.3 Congestion control

The Internet is a shared infrastructure with limited capacity. Due to its increasing popularity over the years, more and more devices are connected to the Internet, increasing the quantity of packets exchanged on the network. Routers forward these packets as quickly as possible, while constrained by either link capacity or the cost of processing them, e.g., updating fields in the IP header. When packets arrive faster than these routers can process them, they remain in the *receiving queue*. If the queue becomes full, the node can decide to drop packets, which will be recovered through retransmissions of the transport protocol. However, packets remaining in the queue are not dropped; instead, their transmission time increases. This phenomenon led to the first *Internet congestion collapse* in October 1986 [Jac88]. The pressure from the numerous devices connected to the Internet saturated the routers' receiving queues, creating *congestion events* in which the packet transmission time became significantly longer than the previously measured round-trip time on the end-hosts. These hosts then marked the packets as lost, triggering spurious retransmissions, which further increased the routers' queues, creating a vicious circle. Measurements at that time showed a goodput up to 1000 times lower during the first congestion collapse. This event led to the development of *congestion control algorithms*, whose objective is to limit the emitter's transmission rate in reaction to these events. End-hosts implement these algorithms at the transport layer, where they access connection statistics (e.g., RTT measurements) and control packet transmission timing. These algorithms require network signaling to detect congestion events.

Explicit Congestion Notification. A first suggested solution used *explicit notification* of congestion events from routers in the network [RJ88]. However, this solution required updating all network nodes to support the feature. Still, this work led to the standardization of *Explicit Congestion Notification* (ECN) [RFC3168]. ECN uses two bits in the IP header to indicate support for the feature and to explicitly notify congestion. Intermediate routers along the transmission path can mark the packet by setting the CE bit to 1 when they detect congestion. The idea behind ECN is to notify end-hosts of congestion events *before* packet losses occur, thereby decreasing the number of retransmissions in the network. The exact timing for marking the CE bit based on buffer queue occupancy is determined by Active Queue Management algorithms. Though we do not detail them here, modern congestion control algorithms leverage ECN, and modern architecture using ECN emerged, such as *Low Latency, Low Losses, Scalable Throughput* (L4S) [RFC9330].

Packet losses as implicit signaling. Instead of relying on ECN, researchers inferred congestion events through implicit signaling using *packet losses* [Jac88]. Ideally, a congestion controller must fulfill both *fairness* and *re-*

silience. Fairness ensures that all competing flows share the limited bandwidth capacity equally, despite differing network conditions (e.g., RTT). Moreover, these algorithms must be *resilient* and adapt to moving network conditions. For example, a steady bandwidth share can be disrupted upon the arrival of a new competing flow. Conversely, if more bandwidth becomes available (e.g., a competing flow finishes the communication), the congestion controller must be able to use this increased available bandwidth. As such, these algorithms aim to maximize bandwidth utilization while avoiding network congestion.

Most congestion controllers operate using a *congestion window*, which represents the number of bytes the emitter is allowed to send in the network while avoiding congestion. This value is computed by the congestion control algorithm based on its view of the network state. Next to it lies the number of *bytes in flight*, i.e., the number of bytes sent to the peer that have not been acknowledged yet. When a packet is acknowledged or lost, the emitter subtracts its length from the number of in-flight bytes, recomputes its congestion window according to its congestion control algorithm, and potentially sends packets if the number of bytes in flight is less than the new computed window. Two families of congestion control algorithms co-exist with different strategies and goals. The primary (and first designed) family relies on loss events to update its window, while the second models the state of the network to maximize bandwidth usage with low latency increase.

1.4.3.1 Loss-based congestion control algorithms

The first congestion control algorithms were loss-based with the initial work from Jacobson [Jac88]. These algorithms assume that congestion events account for the vast majority of packet losses on the Internet. These algorithms split into two phases: *slow-start* and *congestion avoidance* [RFC5681].

Slow-start. The congestion window (*cwnd*) is initially set to a low value to avoid saturating a low-capacity network at the beginning of the connection. The slow-start phase aims to quickly increase this *cwnd* to gain initial insight into available bandwidth. During this phase, the congestion window is roughly doubled every RTT. For every new ACK received from the peer:

$$cwnd = cwnd + \min(N, SMSS), \tag{1.2}$$

where N is the number of bytes previously unacknowledged that are acknowledged in the incoming ACK, and $SMSS$ is the sender's maximum segment size. When the first loss is seen, the congestion window is halved, setting the first value of the *slow-start threshold* ($ssthresh$). If $cwnd < ssthresh$, the algorithm re-enters the slow-start phase; otherwise, it enters *congestion avoidance*.

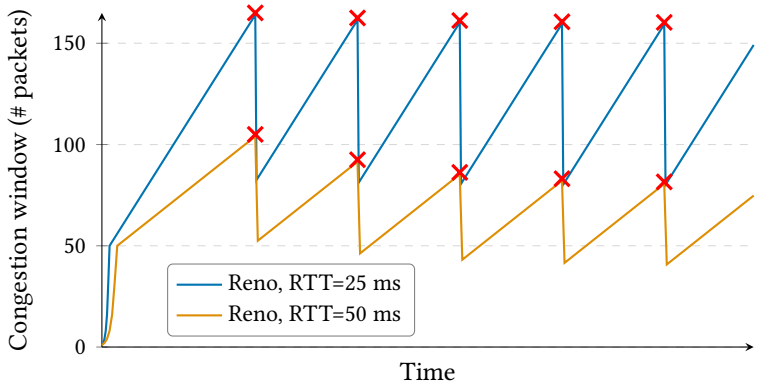


Figure 1.6: Evolution of the congestion window of Reno when varying the RTT.

Congestion Avoidance. This phase aims to carefully probe for additional bandwidth, provided no congestion event is observed, using the *additive-increase-multiplicative-decrease* strategy. In a nutshell, the congestion window is increased by *SMSS* every RTT. When a light congestion event is observed (e.g., 3 consecutive duplicate ACKs), the *cwnd* is halved, and the *ssthresh* is updated to the same value. A retransmission timeout (RTO) expiration is considered a severe congestion event, and the window is set to *SMSS*. Because $cwnd < ssthresh$, the emitter enters the slow-start phase to probe for more bandwidth more quickly.

This algorithm is often referred to as the *Reno* algorithm, in reference to the BSD release that implemented it in 1990 [FF96]. The Reno congestion control algorithm struggled to achieve fairness when two competing flows had significantly different round-trip times, because its *additive-increase* is directly linked to the measured RTT. Figure 1.6 shows how the RTT impacts the evolution of the congestion window of Reno. Moreover, the linear additive increase induces a long delay before the congestion window reaches the value just before the last congestion event in networks with a high bandwidth-delay product (BDP).

To address this, researchers developed the CUBIC [RFC9438] algorithm. CUBIC retains the slow-start and congestion-avoidance phases but introduces two changes that remedy the aforementioned Reno issues. CUBIC uses a cubic function (rather than Reno's linear increase) in the congestion-avoidance phase. Figure 1.7 illustrates an example of the evolution of the congestion window of a CUBIC sender. After a congestion event, CUBIC rapidly increases its congestion window and plateaus at the previous window value that triggered the event. After a slow increase around this value, CUBIC aggressively probes for additional bandwidth in case the network condition

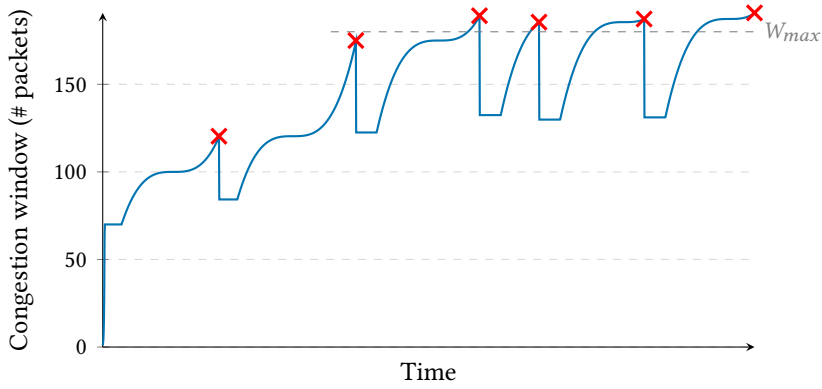


Figure 1.7: Evolution of the congestion window of CUBIC, which is independent of the RTT.

changes. Since 2006, CUBIC is the default congestion control algorithm in Linux distributions [Lin06].

1.4.3.2 Model-based congestion control algorithms

Loss-based algorithms react to losses as a sign of congestion. Depending on the Active Queue Management (AQM) strategy, these losses occur only when the receiving queue on the routers is (almost) full. Even if packets are not dropped, their transmission time can significantly increase due to the additional time they spend in these queues, especially when routers keep large buffers, thereby artificially increasing the connection's delay. This phenomenon is often called *bufferbloat* [GN12]. A first obvious solution is to decrease the buffer size, which may trigger unnecessary retransmissions because routers drop packets more easily.

Google researchers introduced the *Bottleneck Bandwidth and Round-trip propagation time* (BBR) algorithm in 2016 [Car+16], which suggested another strategy to estimate the bandwidth endpoints can use. Since this initial work, two additional versions of BBR have been studied, bringing incremental improvements. BBRv3 [CSB26] is the latest version discussed within the IETF. Instead of solely relying on packet loss, BBR models the network path using recent measurements of the delivery rate, the evolution of the RTT, and packet loss[1]. BBR estimates the bandwidth-delay product (BDP) based on this model, and regulates both its pacing rate (to match the estimated bandwidth) and its in-flight data (to approximate the BDP), targeting maximum link utilization with minimal buffer occupancy. To quickly sample the maximum available bandwidth at the beginning of the transmission, the BBR emitter increases

[1]BBRv1 [Car+17] did not use packet loss to model the network path.

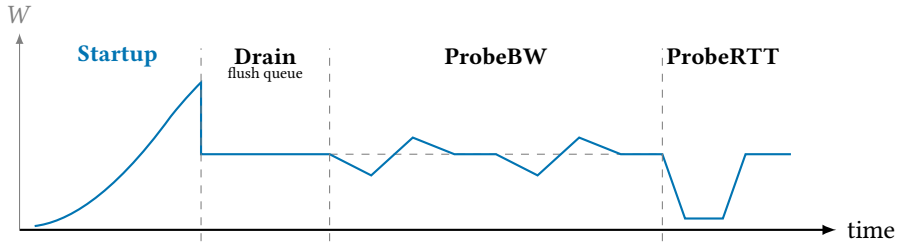


Figure 1.8: Scheme of the BBR congestion control algorithm, which alternates between ProbeBW and ProbeRTT phases to model the link.

its data rate, then reduces it to allow routers to drain their queues after the generated burst of packets. It then alternates between the *ProbeBW* and the *ProbeRTT* phases. Figure 1.8 illustrates BBR's behavior.

ProbeBW. The *Probe Bandwidth* state is the steady state of a BBR emitter. Packets are paced at a rate close to the estimated bottleneck bandwidth, while the amount of in-flight data is bounded around the BDP. During this phase, BBR periodically slightly increases the sending rate above the BDP to probe for additional bandwidth, and then decreases it to drain the buffer's occupancy. Using the ACKs from its peer, the BBR emitter estimates the available bandwidth.

ProbeRTT. Occasionally, BBR significantly reduces its sending rate to avoid contributing to buffer occupancy. With this reduced rate, it samples the RTT. Using the measured RTT and previously measured bandwidth, the algorithm can estimate new BDP samples.

1.4.4 Transport-layer security

During the design of UDP and TCP, the Internet was a research project involving only a few trusted parties, and it did not handle sensitive data. As the project gained popularity and ultimately became a public infrastructure, the need to protect and authenticate the data exchanged increased. The Transport Layer Security (TLS) protocol [RFC2246] was designed to protect the application payload over a TCP byte stream. With TLS, two endpoints negotiate the cryptographic material, such as which encryption algorithm to use, ultimately deriving a set of *TLS* keys that only these two endpoints possess. They then use these keys to encrypt and authenticate the content as *TLS Records*, which are subsequently sent over the TCP connection. The latest version of the protocol is TLS 1.3 [RFC8446]. The main sensitive part of the TLS handshake is to derive the *TLS Secrets*, from which the set of TLS keys is computed. An entity with access to these TLS secrets, although not part of the handshake

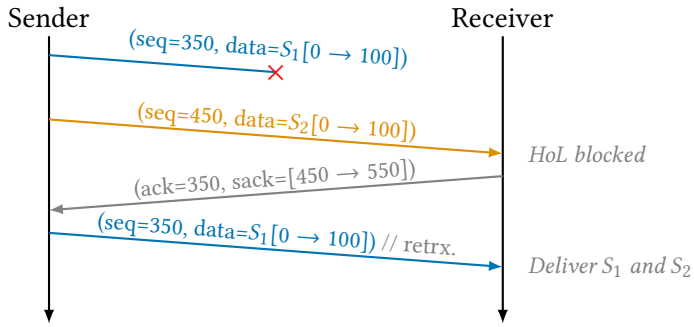


Figure 1.9: Head-of-Line blocking in TCP with HTTP/2. A packet loss in Stream 1 prevents TCP from delivering Stream 2 data to the application, despite it being already received.

itself, can reconstruct the corresponding TLS keys. Let us note that these TLS secrets are derived independently on each endpoint using the HKDF algorithm [RFC5869], ensuring that both endpoints deterministically compute identical key material without exchanging these secrets over the network.

Because TLS is negotiated independently of TCP, the TLS handshake is performed only *after* the TCP three-way handshake completes, so an application using TLS 1.3 requires a total of 2 RTTs before being able to send data; using an older version required 3 RTTs. The Fast-Open extension [Che+14] has been proposed to reduce the cost of connection establishment in TCP, allowing the client to send application data alongside the connection establishment – and, with TLS 1.3, which also introduced the same feature – to a server with a previously established connection, but this extension is hardly deployed.

1.4.5 Shortcomings of TCP

Designed more than 40 years ago, TCP faces shortcomings in today's applications, especially for web content, which typically uses a TCP+TLS+HTTP stack. As discussed above, the sequential TCP-then-TLS handshake adds latency, and the TCP header is forwarded in clear text, allowing external attackers to track an ongoing TCP connection. In addition, TCP faces the *Head-of-Line* (HoL) blocking problem due to its single-stream nature, as illustrated in Figure 1.9. In this example, an HTTP/2 server sends two files in different HTTP streams over TCP. Despite multiplexing enabled in HTTP/2 [RFC9113] (older HTTP versions also suffered from HoL blocking at the application level), a packet loss during the transmission of the first response blocks the second because TCP delivers the byte stream in order.

1.5 The QUIC protocol

QUIC [RFC9000] is a modern transport protocol aiming to solve the shortcomings of TCP and replace it for web content distribution. Initially standing for *Quick UDP Internet Connections*, QUIC was launched by Google in 2012 [QUIC-CHROMIUM; QUICBLOG] as an experimental project to decrease the number of round-trips required to establish a secure connection before forwarding HTTP data. QUIC combines transport and security (through TLS) in the same design, bringing multiple benefits. First, connection establishment is quicker because both the transport connection and TLS sessions are established simultaneously. Second, most parts of the QUIC packets are encrypted, preventing third-party eavesdropping. QUIC also solves head-of-line blocking with stream multiplexing, similarly to the Stream Control Transport Protocol (SCTP) [RFC9260]. QUIC runs on top of UDP, meaning that QUIC packets are sent as UDP datagrams. It allows QUIC to rely on existing SOCK_DGRAM sockets to interact with the kernel. As such, the protocol can be implemented as a user-space library and packaged inside applications. Updating or extending QUIC is much simpler because it only requires a new library version, unlike TCP, which inherently suffers from ossification [Hon+11]. Moreover, relying on UDP ensures that QUIC can traverse most firewalls on the Internet, since this unreliable protocol has co-existed with TCP for more than 40 years.

The success of this first experimental version of QUIC by Google [Lan+17] pushed the IETF to create a working group dedicated to standardizing the protocol in 2016 [QUICWG], leading to QUIC version 1 in 2021 [RFC9000]. As of April 2026, QUIC is used in more than 38 % of web communication across the world [QUIC-W3]. In Belgium, around 70 % of web object fetching uses QUIC [QUIC-LABS], which has been in constant increase since its initial release [Lan+17; QUIC-APNIC; Zir+21].

Frame container. To further facilitate extensibility, QUIC packets work as *frame containers*. The protocol defines a set of *frames*, each with a specific purpose for control or data delivery [RFC9000]. For example, the STREAM frame carries a reliable, ordered stream of data, while the ACK frame contains acknowledgment ranges. Extending QUIC with a new feature is done by adding a new frame type with the desired control logic [De +19; MB24; Mic+22a].

Connection Identifiers. A major difference between TCP and QUIC is how endpoints identify connections. While TCP uses the *four-tuple* (source and destination IP addresses and port numbers), QUIC relies on source and destination *Connection Identifiers* (CID). The primary role of this CID is to remain capable of identifying an ongoing connection despite changes in the underlying network address and/or port, for example, due to NAT rebinding, but servers can also use this CID to embed information, e.g., for load-

balancing [DBH25]. Each endpoint uses its own source Connection ID as the *Source CID* during the handshake and uses the remote endpoint's Connection ID as the *Destination CID*. Because the client does not know the server's Source CID when it sends its first QUIC packet, it initially chooses a Destination CID, which the server will change. Connection Identifiers are encoded on up to 20 bytes.

1.5.1 Connection establishment

Similar to TCP, QUIC endpoints establish a connection through a handshake to ensure bidirectional reachability and to negotiate the use of extensions and cryptographic algorithms for secure data transfer. By embedding TLS 1.3 into its design, QUIC reduces the two round-trips from TCP+TLS to a single one by simultaneously establishing the transport connection and the security session. Similarly to TCP Fast-Open, QUIC's design includes the 0-RTT feature, allowing a client to send application data during connection establishment, if state from a previous connection exists. The feature is implemented in most QUIC stacks [See25]. Finally, *transport parameter* exchange also occurs during the connection handshake. Transport parameters are protocol options that both endpoints negotiate, e.g., to discuss initial flow-control values or indicate support for features such as Connection Migration.

1.5.2 The QUIC packet

QUIC uses two packet formats, depending on the connection state. During connection establishment, endpoints use the *Long header format*, i.e., the header explicitly contains all the required information to create state on both endpoints. Once the handshake is complete, endpoints use *Short headers* (also called *1-RTT* packets) to reduce overhead.

The Long Header has four different types, depending on the state of the connection handshake. *Initial* packets contain the CRYPTO frames for the TLS ClientHello/ServerHello, and are protected using the Destination CID chosen by the client; as such, the traffic is not completely protected because an external observer may decrypt the packet content. Then, the *Handshake* packets carry the remaining TLS handshake data and are protected using the keys derived during the first phase. The client uses the *0-RTT* packet to send early data if a previous connection to the same server exists. Finally, the server can reply to the client's first *Initial* with a *Retry* packet containing a token that the client must echo to verify that it is not spoofing an IP address.

The Short Header (1-RTT) is used for the remainder of the connection once the handshake is complete on both sides. This is illustrated by Figure 1.10. The header contains only a few unencrypted fields. The remaining control information is either contained in *frames* (i.e., payload of the QUIC

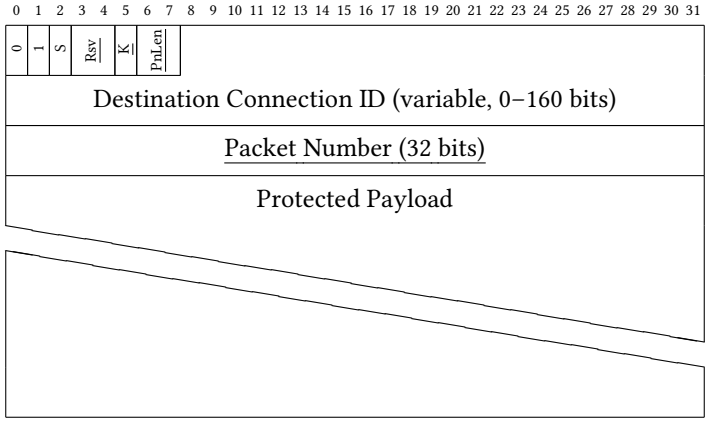


Figure 1.10: QUIC 1-RTT (short header) packet format (RFC 9000). `S` = spin bit, `Rsv` = reserved, `K` = key phase, `PnLen` = packet number length. In this example, the packet number is 32 bits long for readability, but could be shorter (8-32 bits).

packet) or part of the encrypted header. Underlined values are part of the encrypted header.

- The *Header Form* indicates whether this is a short (0) or long (1) header;
- The *Fixed Bit* is always set to 1, allowing quick identification of non-QUIC packets;
- The *Spin Bit* allows passive monitoring of the round-trip time (RTT) from observation points along the path between the two hosts. The server echoes the received value, while the client spins the bit every RTT;
- The *Reserved Bits* must be set to 0;
- The *Key Phase* indicates the key phase, allowing the recipient to select the appropriate protection keys to decrypt the packet [RFC9001];
- The *Packet Number Length* indicates the length of the *Packet Number* in bytes, minus one;
- The *Destination Connection ID*, as presented earlier, allows the recipient to match the packet to an existing QUIC connection;
- Finally, the *Packet Number* is encoded between 1 and 4 bytes, and contains the least significant bits of the 64-bit packet number of the QUIC packet.

The remainder of the packet contains the encrypted QUIC Payload.

1.5.3 Packet number and packet number spaces

QUIC uniquely identifies packets using a monotonically increasing *packet number*, independent of the carried frames. It contrasts with TCP's sequence number, which is the position of the first byte of the segment's payload in the byte stream. When a QUIC packet carrying reliable frames (such as a STREAM frame) is lost, the content of the frames is sent in a new packet *with increased packet number*. This way of identifying packets has several benefits over TCP, as it allows endpoints to disambiguate lost packets from retransmissions, enabling more accurate and efficient loss recovery and RTT estimation. Nothing constrains a QUIC packet emitter to use contiguous packet numbers, as we will further analyze in Chapter 5.

Packet number spaces. QUIC uses distinct packet number spaces for each different packet type. As each type uses a distinct set of encryption keys, it ensures that acknowledgments of packets with one key do not cause spurious retransmissions of packets encrypted with another set. This mechanism implies that the packet numbers can be reused across different packet types, but not within the same packet type.

1.5.4 QUIC frames

The QUIC payload consists of one or more *frames*, which are logical entities used for both control and data delivery. These frames are encrypted and authenticated [RFC9001], preventing external observers from inspecting their content. Unlike TCP, where acknowledgments and flow-control updates are forwarded in clear text, this information is securely exchanged within dedicated frames. Each frame has its own type, encoded as a *variable-length integer*. We review the most relevant frames of QUIC version 1 in the context of this thesis.

- PING: the role of this frame is to trigger an ACK frame from the recipient. It can be used to keep the connection alive if no application data needs to be sent.

- ACK: it contains ranges of packet numbers received by the emitter of this frame. Similarly to TCP SACKs, the emitter can advertise multiple ranges of acknowledged packets, but the number of ranges is limited to QUIC's packet size rather than the TCP option length. The ACK frame only acknowledges packets from the same *packet number space*. Because the ACK frame is used to estimate the RTT of the connection, delaying sending such ACK frame could inflate such RTT estimation. To prevent this inflation, the ACK frame contains the *ACK Delay*, indicating the time the emitter waited between reception of the *last* acknowledged packet and the emission of the frame. The ACK frame also

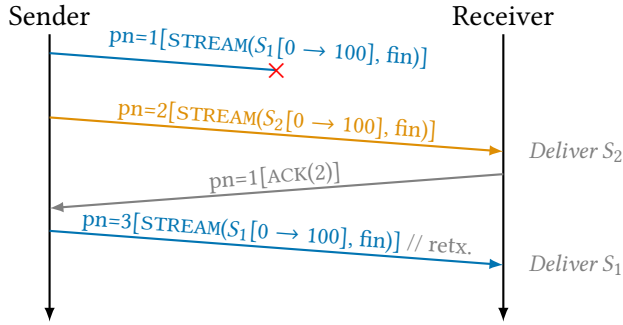


Figure 1.11: QUIC's stream multiplexing prevents the Head-of-Line blocking problem. Stream S_2 is delivered even if S_1 is lost. The content of S_1 is retransmitted in a new QUIC packet with increased number.

contains ECN counts. An endpoint must send ACK frames upon receiving *ack-eliciting* packets.

- STREAM: this frame carries reliable, ordered byte stream data. It contains the *Stream ID*, which enables multiplexing, as well as the payload offset and length for this frame. We detail the principles of QUIC streams in the subsequent section.

- DATAGRAM: the unreliable datagram QUIC extension [RFC9221] defines this frame as a way to send unreliable datagrams in a QUIC connection, exposing the same service as UDP. DATAGRAM frames are also subject to congestion control limits.

- NEW_CONNECTION_ID: an endpoint sends this frame to issue new source Connection IDs that can be used during the connection.

- PATH_CHALLENGE and PATH_RESPONSE: endpoints send these two frames to verify the reachability of a new network path.

1.5.5 QUIC Streams

QUIC solves the Head-of-Line blocking problem by introducing stream multiplexing, as illustrated in Figure 1.11. If packet 1 carrying a STREAM frame with identifier S_1 is lost, only subsequent data from the same stream will be blocked, which is expected because streams provide an ordered byte sequence. However, the stream with identifier S_2 can be processed by the application even if the previous packet is lost, since both streams are independent. The two lowest-order bits of the stream identifier indicate its *stream type*: streams can either be unidirectional or bidirectional, and client or server-

initiated. For example, a unidirectional, server-initiated stream has its lowest bits set to `0b11`, i.e., streams 3, 7, 11, ...

1.5.6 Reliability mechanism

QUIC implements its own reliability mechanism to ensure the correct delivery of its streams. The protocol leverages recent advancements in TCP loss recovery mechanisms, namely Fast Retransmit [RFC5681], SACK [RFC6675], and RACK-TLP [RFC8985]. QUIC uses acknowledgments to detect packet losses, and probe timeouts to trigger such acknowledgments [RFC9002]. In a nutshell, a QUIC sender considers a packet as lost if it meets the two following conditions:

- This packet is in flight but unacknowledged, and at least a packet with a higher packet number is acknowledged;
- This packet was sent at least $kPacketThreshold$ packets before an acknowledged packet, *or* it was sent more than $tTimeThreshold$ in the past.

QUIC recovery's specification recommends setting $kPacketThreshold = 3$ and potentially increasing it if larger network reordering is detected to avoid spurious retransmissions. The time-based threshold, $tTimeThreshold$, is computed using the RTT estimation:

$$tTimeThreshold = \frac{9}{8} * \max(smoothed\_rtt, latest\_rtt), \tag{1.3}$$

where $smoothed\_rtt$ and $latest\_rtt$ are, respectively, the smoothed RTT and the last RTT sample, and assuming a minimum RTT of 1 ms.

1.5.7 Applications relying on QUIC

HTTP/3 [RFC9114] is the most prevalent application protocol running on top of QUIC, as it natively supports the protocol's stream multiplexing and benefits from its integrated security features. The IETF discusses new extensions atop QUIC and HTTP/3, such as WebTransport [FKV26], Media over QUIC [Nan+26] and MASQUE [RFC9298]. Other protocols, such as the Domain Name System (DNS), have been standardized on top of QUIC [RFC9250].

QUIC Implementations and Interop Runner. More than twenty open-source QUIC implementations co-exist [IET25], in several programming languages. The QUIC interop runner [SI20; See25] is an initiative to provide an open-source, free, and fully automated tool to assess the interoperability of QUIC stacks in different scenarios. Seventeen implementations currently participate in the QUIC interop runner.

1.6 Managing multiple paths for a QUIC connection

Inspired by the standardization of Multipath TCP (MPTCP) [RFC8684] as the IETF leaned toward QUIC [RFC9000], researchers investigated extending the protocol to manage multiple paths within a single QUIC connection [DB17]. While QUIC itself has significantly changed since then, this work led to the adoption of this extension for standardization in 2022 [Liu+26]. At the time of writing this thesis, *Managing multiple paths for a QUIC Connection* is becoming an IETF standard. In this section, we present the extension, highlighting the key design ideas we leverage throughout this thesis. For simplicity, we refer to *Managing multiple paths for a QUIC Connection* as *Multipath QUIC*.

Multipath use-cases. By managing multiple paths within the same connection, transport protocols expose additional features to applications. Paths can use different interfaces, enabling, for example, a mobile device to use its Wi-Fi and cellular interfaces simultaneously. This combination enables two possibilities initially discussed with MPTCP [RFC8684]:

1. Handle path failover: consider a mobile device moving away from home. It establishes a transport connection to an IP address associated with its Wi-Fi interface, then opens a second path from the IP address of its cellular interface. When the user loses Wi-Fi connectivity, the transport connection seamlessly falls back to the second path using cellular.

2. Path aggregation: the application can leverage these two paths to send and receive data simultaneously. It potentially increases the cumulated bandwidth as it is the sum of the individual bandwidths of the two paths.

QUIC's specification [RFC9000] includes the *Connection Migration* feature, enabling the client to change the network path without breaking the connection, thereby handling path failover without requiring multiple-path management. Still, the IETF pushes forward with this Multipath extension, as new use-cases leveraging the modern design of QUIC have emerged [RHB18; MK26]. Only the client can initiate the creation of new paths.

Extending QUIC path for multipath. QUIC [RFC9000] uses a single active path at a time. To enable multiple paths to be active simultaneously, Multipath QUIC introduces the *path ID*, a monotonically increasing integer, to identify each path. The first path of a QUIC connection, which is established during the connection handshake, has a path ID of 0. Similar to a single-path QUIC connection, which maintains a pool of Connection IDs to use while a single CID is active at a time, each path in Multipath QUIC maintains a set of CIDs to use during its lifetime. Identifying paths by path ID allows the reuse of the same four-tuple from the underlying network path across different QUIC logical paths.

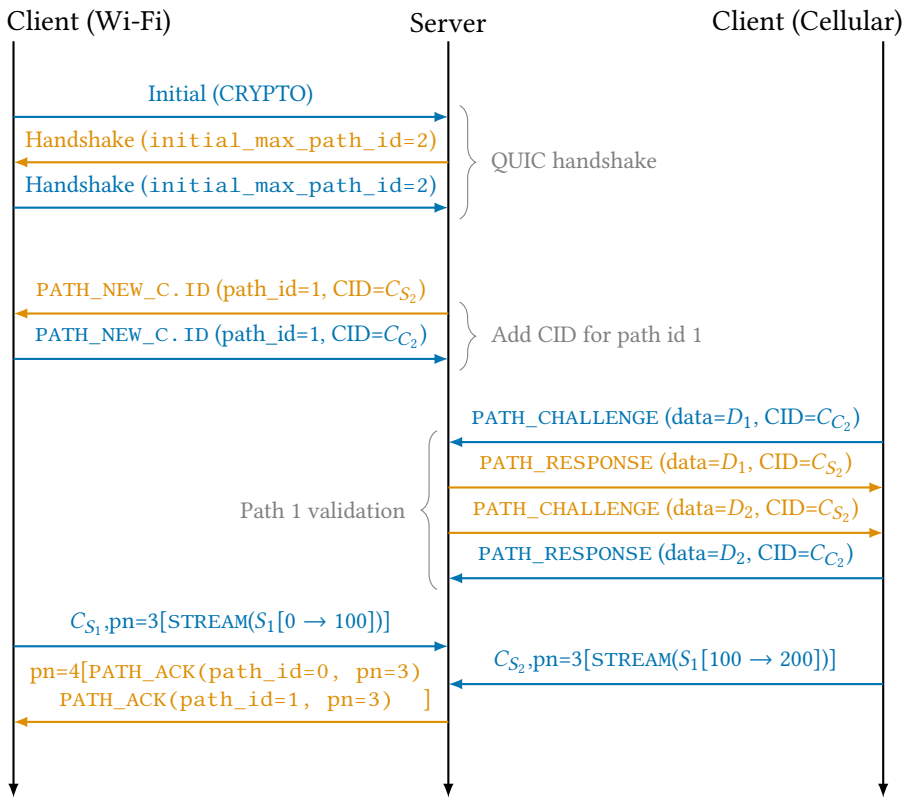


Figure 1.12: Example of negotiation of Multipath QUIC [Liu+26] and creation of an additional path. We assume that the issued Destination Connection IDs used for the first path are C_{C_1} and C_{S_1}, respectively, for the client and the server. In this example, the server sends its PATH_ACK frames on the first path.

1.6.1 Creating new paths

Endpoints negotiate multipath via the `initial_max_path_id` transport parameter, which indicates the maximum path identifier they are willing to initially support. Multipath can only be negotiated if the Connection Migration feature is enabled on both hosts. To create a new path, both endpoints must issue at least one new Connection ID for the new path with an increased path identifier. They exchange these Connection IDs through the PATH_NEW_CONNECTION_ID frame, extending the NEW_CONNECTION_ID by including the associated path ID for which this CID is issued. Figure 1.12 provides an example of creating a new path in Multipath QUIC starting from the handshake.

Path validation is a mechanism introduced in QUIC [RFC9000] to sup-

port its Connection Migration feature and ensure the bidirectional reachability of a new path during an ongoing connection. The client initiates a PATH_CHALLENGE frame containing random 8-byte data, which the server echoes through the PATH_RESPONSE using the same path. To check bidirectional reachability, the server must perform the same challenge-response procedure in reverse. For Multipath QUIC, the packets containing these frames are sent with Connection IDs issued with the path ID of the new path to validate, allowing hosts to correctly identify that they belong to the validation of a *new* additional path.

1.6.2 Managing multiple paths

Now that the connection can manage multiple paths simultaneously, Multipath extends QUIC in several ways.

Packet number spaces. Each Multipath QUIC path uses its own *packet number space*, meaning that packet numbers are completely independent across paths. Compared to a single packet number space shared across all paths, separating them reduces complexity and can even improve delivery times on heterogeneous paths [De 22].

Packet protection. Multipath QUIC uses the same set of TLS keys across all paths. This set is derived at connection establishment. Because the packet number is used when creating the nonce to encrypt the packet header, using multiple packet number spaces may result in the same nonce computation across different paths. To avoid this security issue, Multipath QUIC further uses the path ID when computing the nonce. Because path IDs are unique per path within a connection, it ensures that nonces are not repeated.

Congestion control. Each path has its own congestion control state because, e.g., Wi-Fi and cellular networks may have completely different network quality. Similarly, when a new path is established, hosts must use the initial congestion-control parameters and not infer anything about the congestion state of the new path from older ones. This may raise a fairness issue when two or more QUIC paths share the same underlying network path, since other connections may have to share their bandwidth with more entities on the same connection. Coupled congestion control algorithms [RFC6356] provide additional knowledge to the congestion state estimation.

1.6.3 Sending QUIC packets over multiple paths

When two or more paths are used simultaneously, they can be combined to transfer application data. Hosts are then responsible for choosing which path to use to convey data. This is the role of *Packet Schedulers*. Multiple packet schedulers have been studied in the literature, initially for MPTCP [Paa+14] and later for Multipath QUIC [DB17; RHB18; MK26]. While Multipath QUIC

paths use separate packet number spaces and congestion control states, they share the same stream state. Figure 1.12 illustrates this process: different pieces of streams (using the STREAM frame) can be sent through different paths. This example highlights the benefit of QUIC in separating path logic from application data handling.

The simplest scheduler is the *Round-Robin*, where the host alternates between the two paths to send data. Though simple, it risks blocking the stream if one path offers significantly poorer quality (e.g., lower bandwidth, higher delay) than another. Another well-known scheduler is the *Lowest-RTT*, which primarily sends packets to the path with the lowest RTT. As we will see in this thesis, Multipath QUIC allows us to define more original schedulers, such as only using the second path to retransmit reliable frames that have been lost on the primary.

Acknowledging Multipath QUIC packets. The last important piece brought by Multipath QUIC is the way hosts acknowledge packets. The ACK frame is extended with the PATH_ACK by adding the path ID of the path that conveyed the packets being acknowledged by this PATH_ACK frame. This mechanism will be heavily used in this thesis, as it allows a host to acknowledge packets received on a path by sending a PATH_ACK over *another* path, as illustrated by Figure 1.12, where packets from both paths are acknowledged through a PATH_ACK frame sent over the path using the client's Wi-Fi interface.

1.7 Forward Erasure Correction

Forward Erasure Correction (FEC) in network communication is a mechanism in which an emitter sends redundant data, allowing the peer to recover from packet losses without retransmissions, which take up to an entire RTT to detect packet erasure. FEC is usually referred to as Forward *Error* Correction, i.e., the set of techniques adding redundancy to recover from transmission errors. In this thesis, we use the term Forward *Erasure* Correction, a subset of *Error* Correction that focuses on adding redundancy to recover from packet losses.

FEC operates on *symbols*, which are fixed-size units of data. A FEC encoder takes k *source symbols* and produces r *coded symbols*, such that a receiver collecting any k out of $k + r$ symbols can reconstruct the original k source symbols. The ratio $\frac{k}{k+r}$ is called the *code rate*, and characterizes the proportion of useful data among all transmitted symbols. We call *FEC scheme* the algorithm used to generate the $k + r$ symbols and to recover from the packet losses.

The simplest FEC scheme is the bitwise *XOR* scheme, which generates $r = 1$ additional coded symbol based on the k input source symbols, i.e., a sin-

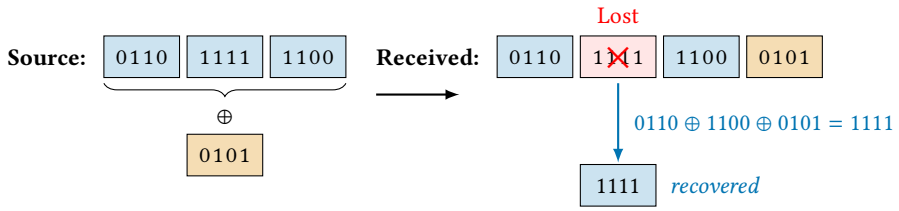


Figure 1.13: Example of Forward Erasure Correction with the XOR scheme.

gle redundant symbol. Figure 1.13 shows an example of the XOR scheme protecting 3 symbols of 4 bits each, and how the recipient recovers the loss of the second symbol. While computationally efficient, the XOR scheme can only recover a single loss per block of k symbols. More complex coding schemes aim to generate multiple additional symbols from the same set of source symbols, thereby increasing recovery capabilities in lossy communication. We call the *encoder* the entity that generates the coded symbols, and the *decoder* the entity that decodes them, potentially recovering from packet losses.

1.7.1 Systematic codes

FEC schemes take as input k source symbols and produce $k + r$ coded symbols, making any subset of k coded symbols sufficient to recover from potential losses to decode the k initial source symbols. Another strategy is to rely on *systematic code schemes*, which do not modify the input source symbols, and produce r additional *repair* symbols. This strategy has multiple advantages, especially regarding latency-sensitive applications. First, because the source symbols are sent *as is*, the encoder can forward them before the repair symbols are generated. Conversely, non-systematic codes add latency on the decoder, as it must wait to receive k coded symbols in order to decode the original source symbols. With systematic codes, the decoder can forward the source symbols as soon as it receives them and use the additional repair packets to recover from the lost source symbols. Second, a more complex FEC scheme than XOR requires more mathematical operations on the symbols. Generating only r repair symbols with systematic codes instead of $k + r$ with non-systematic ones reduces the CPU load on the encoder and decoder. The XOR scheme is a simple example of a systematic code scheme. Because this thesis only relies on systematic codes, the remainder of this section assumes the consistent use of systematic codes.

1.7.2 Block and fountain codes

Block codes operate on a fixed number of source symbols k, generating additional r repair symbols, with a fixed code rate of $\frac{k}{k+r}$. The encoder generates these r repair symbols after seeing the k source symbols it protects. Thus, the r additional symbols protect the same set of source symbols. Block codes can be *Maximum Distance Separable* (MDS), an important property in coding theory stemming from the optimality of the Hamming distance: any subset of k out of the $k + r$ symbols (independently of the mix between source and repair symbols in the context of systematic codes) is always sufficient to recover the k original source symbols. The most well-known MDS block code is the *Reed-Solomon* scheme [RS60].

Fountain codes, also called *rateless* codes, operate without a fixed rate, i.e., without a fixed number of additional symbols per block of k source symbols [Bye+98]. Their advantage over block codes is that they allow for dynamically generating more/fewer repair symbols per k source symbols. Examples of fountain codes include Tornado [Bye+98] and Raptor codes [Sho06]. Fountain codes are not always Maximum Distance Separable, as it heavily depends on the number of generated repair symbols by sets of k source symbols. We provide a more precise explanation of Random Linear Codes (RLC), which we use in this thesis.

1.7.3 Random Linear Codes

Random Linear Codes (RLC) enable rateless coding by generating repair symbols as a linear combination of source symbols using *pseudo-random* coefficients. For k source symbols $S_1, S_2, \ldots, S_k$, RLC generates r repair symbols $R_1, R_2, \ldots, R_r$:

$$\begin{cases} R_1 = c_{1,1}S_1 + c_{1,2}S_2 + \cdots + c_{1,k}S_k, \\ R_2 = c_{2,1}S_1 + c_{2,2}S_2 + \cdots + c_{2,k}S_k, \\ \cdots, \\ R_r = c_{r,1}S_1 + c_{r,2}S_2 + \cdots + c_{r,k}S_k. \end{cases} \tag{1.4}$$

The coefficients $c_{i,j}$ in Equation 1.4 are pseudo-randomly generated. RLC thus generates $k \times r$ such coefficients. Adding a repair symbol only requires generating k new coefficients and computing $R_{r+1} = \sum_{j=1}^{k} c_{r+1,j}S_j$.

If the decoder receives any k symbols out of the $k + r$ source and repair symbols sent by the encoder, it can recover missing source symbols by solving a linear system of equations, where the unknowns are missing source symbols, the coefficients are the corresponding $c_{i,j}$ and the constants are composed of the received source and repair symbols. As an example, let us con-

sider that $k = 4$ and $r = 2$. Equation 1.4 becomes:

$$\begin{cases} R_1 = c_{1,1}S_1 + c_{1,2}S_2 + c_{1,3}S_3 + c_{1,4}S_4, \\ R_2 = c_{2,1}S_1 + c_{2,2}S_2 + c_{2,3}S_3 + c_{2,4}S_4, \end{cases} \tag{1.5}$$

Let us now consider that source symbols S_2 and S_3 have been erased. The decoder builds the following system:

$$\begin{cases} c_{1,2}S_2 + c_{1,4}S_4 = R_1 - c_{1,1}S_1 - c_{1,3}S_3 \\ c_{2,2}S_2 + c_{2,4}S_4 = R_2 - c_{2,1}S_1 - c_{2,3}S_3 \end{cases} \tag{1.6}$$

We can write the Equation as a matrix product:

$$\begin{bmatrix} c_{12} & c_{14} \\ c_{22} & c_{24} \end{bmatrix} \begin{bmatrix} S_2 \\ S_4 \end{bmatrix} = \begin{bmatrix} R_1 - c_{1,1}S_1 - c_{1,3}S_3 \\ R_2 - c_{2,1}S_1 - c_{2,3}S_3 \end{bmatrix} \tag{1.7}$$

Let us recall that the right-hand side of the system is known, since we assume that the decoder received S_1, S_3, R_1, and R_2. The linear system from Equation 1.7 can be solved if the matrix is of full rank, yielding the recovery of S_2 and S_4. The challenge with RLC is to generate the coefficients so as to maximize the probability of obtaining a full-rank system.

1.7.4 Block and convolutional windows

We call *window* the span of source symbols being protected by one or multiple repair symbols. FEC schemes such as Reed-Solomon usually operate on block windows, while fountain codes like RLC can also operate on *convolutional* ones.

In a *block-window* mode, the repair symbols are generated and forwarded in batch *after* the source symbols they protect, as illustrated in Figure 1.14. If S_4 and S_5 are lost, the decoder must wait for the reception of R_1 and R_2 to recover them both, which adds latency for the application. With a *convolutional* (or *sliding*) window, repair symbols are generated over the current window of source symbols. When new symbols are to be protected, they enter at the end of the window, sliding it by one element. Repair symbols can be generated sequentially on overlapping windows, as illustrated by Figure 1.14. Again, considering the erasure of S_4 and S_5, the decoder recovers S_4 directly after the reception of R_1, and subsequently recovers S_5 after the reception of R_2.

1.7.5 Network packet as source symbols

This section leverages Forward Erasure Correction techniques to protect network packets as source symbols. The first example from Figure 1.13 operates on bits, while network packets weigh up to $\sim$ 1500 B on the Internet. FEC

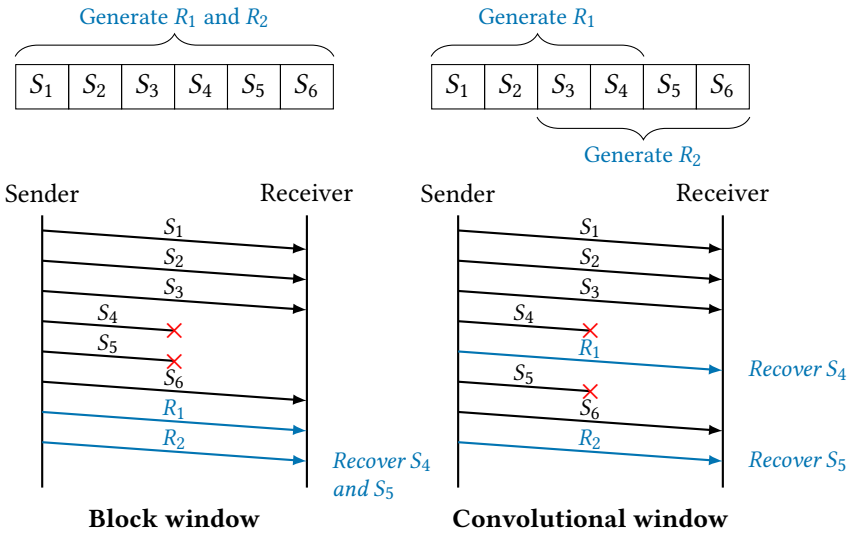


Figure 1.14: Per-block (left) and convolutional windows (right).

techniques can be applied at the bit-level or packet-level, i.e., they can recover from bit flipping or from complete packet erasure. Here, we investigate the use of FEC to recover from complete packet erasure. Bit-level protection, or at least *verification*, is performed by transport-layer protocols such as UDP and TCP, which both include a checksum of their respective header, the pseudo-IP header, and the payload. On the other hand, routers usually drop entire packets when they face congestion or due to medium imperfection. As such, we focus on *packet-level* protection, i.e., source symbols are composed of packets.

Equation (1.7) shows that we can recover from packet losses by solving linear systems of equations. However, the bytes that compose network packets are integers constrained to the natural numbers, whereas techniques such as Gaussian Elimination on computer systems may introduce floating-point imprecision. To perform such complex mathematical operations on network packets, the standard approach is to rely on Galois Fields, which exclusively operate on an integer space. Because packets are composed of bytes, we use the Galois Field $GF(2^8)$. In this field, addition and subtraction are performed using XOR operations, whereas multiplication and division require more complex successive operations.

1.8 Multicast

The rise in popularity of the Internet motivated researchers to design more efficient ways to distribute information to receivers. Protocols we have de-

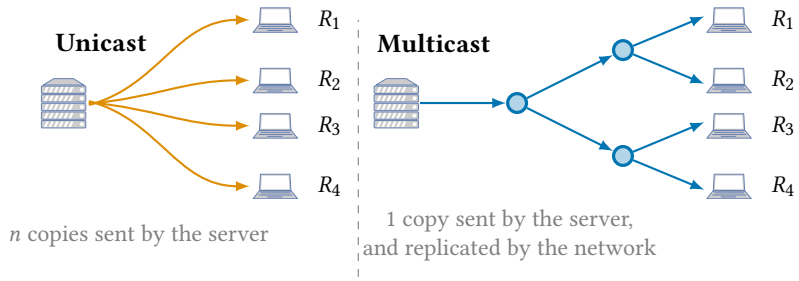


Figure 1.15: Example of unicast and multicast.

tailed so far use *unicast* forwarding: a single source sends packets to a single receiver. This communication method works well for web content, e.g., a client requesting a specific page from a specific server. However, use-cases in the 90's emerged in which *multiple* receivers request *the same* content *simultaneously*, such as to participate remotely in IETF and SIGCOMM meetings. Considering n simultaneous clients in such scenarios, using unicast protocols leads to inefficient network resource usage and additional work for the server, as illustrated in Figure 1.15. We call this communication method *one-to-many*, in opposition to the *one-to-one* method we described until now.

A much more efficient way to distribute this duplicate content is to send the data once and let the network replicate it to all receivers. It led to the development of *multicast* protocols, in which the core idea is that *at most* one copy of each packet traverses every link in the network, enabled by intelligent replication. This mechanism of duplicating packets only when needed forms a *multicast tree*, rooted at the source, with receivers acting as leaves.

1.8.1 IP Multicast

IP Multicast is the first major initiative to improve the efficiency of *one-to-many* communication, inspired by Dr. Deering's work [Dee88; RFC1112] and based on the Internet Protocol (IP). An IP Multicast group is identified with a *group IP address* that receivers can listen to. IP Multicast reserves the IPv4 224.0.0.0/4 prefix, and ff00::/8 prefix for IPv6. In opposition to unicast, where the sender uses the receiver's IP address to send packets, a multicast sender uses the multicast group's IP address as the destination address.

IGMP/MLD. Receivers advertise their willingness to listen to a specific multicast group to their access router with the Internet Group Management Protocol (IGMP) [RFC1112; RFC2236] in IPv4 and the Multicast Listener Discovery (MLD) [RFC2710] in IPv6. The access router then propagates the membership information upstream through the multicast routing infrastructure.

Protocol-Independent Multicast (PIM) is the most widely used pro-

tocol to build the multicast tree, available in multiple flavors. The initial multicast design relied on the *Any-Source Multicast* (ASM) model, in which a multicast group is uniquely identified by its group IP address. PIM Dense-Mode (PIM-DM) [RFC3973] uses a flood-and-prune algorithm to build the multicast distribution tree. Multicast packets are first *flooded* in the network, and routers without active membership for the multicast group send *prune* messages to prevent further packet arrivals. PIM Sparse Mode (PIM-SM) [RFC7761] defines the concept of *Rendezvous Point* (RP), a router chosen in advance and known to all routers in the network, that operates as the root of the multicast trees of the network. The first-hop router of the source sending packets to a specific multicast group must first tunnel them towards the RP, which then starts sending packets along the tree.

Upon reception of an IGMP/MLD request from an end-host, the router forwards PIM Join messages for the specific multicast group IP *toward* the Rendezvous Point, using the underlying multicast routing tables to determine the best path to reach the RP. These PIM Join messages are propagated hop-by-hop, creating state on each intermediate router that indicates: "For this multicast group IP address, the router will forward each packet to the interfaces from which it received a PIM Join". If a router receives a PIM Join for a group for which it already has state, it does not propagate the message further. Instead, it adds the interface from which this later message was received to the list of outgoing interfaces for this group. This mechanism creates a *reverse shortest-path* multicast tree toward the RP, as it is built from the leaves. Because PIM uses information from the underlying routing protocol, it is protocol-independent when building this tree.

The MBONE [Eri94] was the first large-scale deployment of IP Multicast on the Internet. It was initiated in 1992 as an experimental project to support multi-point audio and video conferencing, supporting multiple protocols, including VIC [MJ95] and VAT.

The ASM model suffered from several issues. First, using a Rendezvous Point means that *all* multicast trees start from the same place, limiting the network's scalability because the RP becomes a bottleneck when numerous trees are present. Second, the model prevents source filtering, meaning that any potentially malicious device could send data to any group. Third, address collisions arise when multiple sources distributing different content use the same multicast group IP by accident. These issues led to the design of Source-Specific Multicast (SSM) [RFC4607] and the deprecation of ASM on the Internet [RFC8815].

Source-Specific Multicast (SSM) identifies a multicast group through the group IP G and the source's IP address S, thus forming the (S, G) tuple. SSM addresses the aforementioned issues by allowing PIM to propagate JOIN messages directly toward the source, avoiding the need for a Rendezvous

Point. The remainder of this thesis focuses on Multicast SSM.

1.8.2 Multicast Transport

Applications primarily use UDP as the transport layer for multicast. TCP's design does not extend to one-to-many connections, making multicast communication initially unreliable. For more than two decades, extensive research has been conducted to design multicast transport protocols [LG96; Obr02; LP96] that offer the same service and agility as their unicast equivalents, such as TCP.

One-to-many transport challenges. The one-to-many nature of multicast necessitates a different paradigm than what was designed for unicast protocols. As a comparative note, one can imagine the potential frustration during primary/high school lectures. A single teacher must adapt their information rate to the whole classroom, which is composed of students with different levels of understanding, as opposed to private lessons where the teacher adjusts the explanation to a specific student. This analogy transposes to multicast transport naturally. Multicast congestion control algorithms must handle heterogeneous receiver sets, which forces them to limit the sending rate to the slowest client, thereby impacting communication for the other receivers [ML09]. For example, PGMCC [Riz00] elects a single member of the group to infer the sending rate of the source. Another possibility is to use multiple groups, each with a different bit rate, to accommodate slower receivers without impacting those with higher available bandwidth [KB03].

The TCP-Friendly Multicast Congestion Control (TFMCC) [RFC4654], initially proposed in [WH01], defines a congestion control for single-rate multicast delivery, based on receiver feedback, and adapting to the slowest receiver in the group. TFMCC uses a simplified version of Reno's algorithm, by computing the estimated rate X as:

$$X = \frac{s}{RTT \times (\sqrt{2p/3} + (12 \times \sqrt{3p/8} \times p \times (1 + 32p^2)))}, \tag{1.8}$$

with p the loss rate, RTT the RTT in seconds and s the packet size in bits.

Another challenge arises with one-to-many reliable communication. Similarly to congestion control, a receiver exposed to numerous losses may block the remainder of the group, resulting in degraded communication quality. Moreover, reliability is ensured through acknowledgments, leading to the *ACK implosion* problem in multicast, since a single data packet sent by the source triggers an acknowledgment from each receiver, potentially overwhelming the source. Research explored several ways to mitigate the ACK implosion problem, including hierarchical acknowledgments and negative ACK (NACK). RMTP [Pau+97] clusters receivers in sub-groups, and elects a designated receiver per group that is responsible for collecting acknowledgments

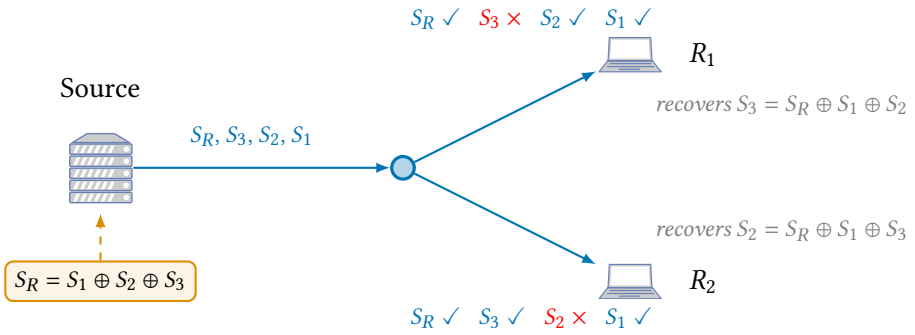


Figure 1.16: Example of Forward Erasure Correction using the XOR, generating a single repair symbol S_R to protect three source symbols S_1, S_2, and S_3.

from its members and sending an aggregated ACK to the source. Pragmatic General Multicast (PGM) [Gem+03] combines hierarchical acknowledgments, NACKs, and NACK suppression to scale to large receiver groups.

NACK and suppression. A common technique to scale while limiting the impact of the ACK implosion problem is to use negative acknowledgments (NACK). Instead of acknowledging received packets, receivers report only gaps in the observed sequence, if any. As such, in perfect condition without packet losses, receivers do not send any feedback to the source. Another issue arising from NACK is the *NACK implosion* problem, which occurs when numerous receivers observe the same gaps in the packet sequence. These receivers all send the same report for the same gap simultaneously, overwhelming the source. Researchers proposed *NACK suppression* to mitigate the impact of NACK implosion: instead of directly sending a NACK, receivers start a random timer before sending the message [AM02]. The first NACK the source receives is echoed to all receivers via multicast. Receivers getting the echoed NACK for the same gap cancel their own NACK. In the ideal case, a single NACK returns to the source.

Forward Erasure Correction (FEC) suits multicast communication, as a single repair packet can recover *distinct* losses on different clients [BLM02], as illustrated in Figure 1.16. In this example, client R_1 received symbols 1 and 2, while R_2 received 1 and 3; both received the repair symbol S_R. Even using a simple XOR algorithm, a single repair packet sent on multicast allows both receivers to recover their losses. Multiple multicast transport protocols rely on FEC, either proactively [RKT98] or upon loss detection [Gem+03].

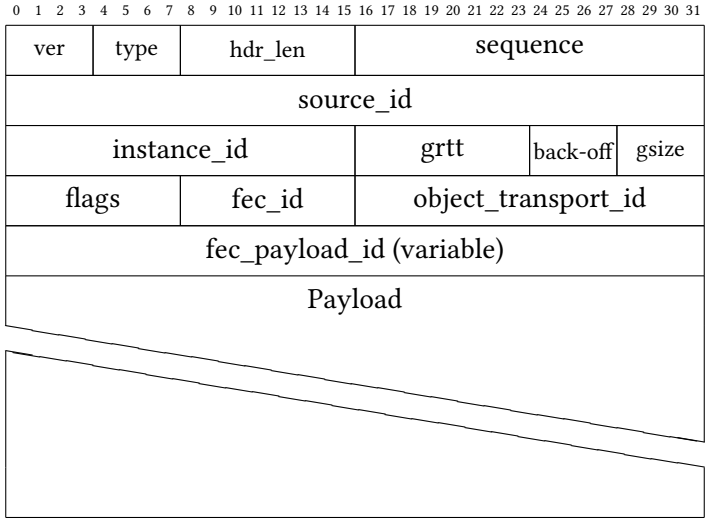


Figure 1.17: NORM NORM_DATA message format.

1.8.3 NACK-Oriented Reliable Multicast

The NACK-Oriented Reliable Multicast (NORM) protocol [RFC5740] is an IETF standard since 2009. NORM uses multicast above UDP to send data to receivers, assuming a working underlying multicast network. Figure 1.17 represents NORM's header for application data forwarding.

The first eight bytes of the header are generic to all NORM messages, while the remaining bytes are specific to NORM_DATA messages, which convey application data. NORM combines NACK with echo/suppression to limit (N)ACK implosion, and uses Forward Erasure Correction to recover from packet losses instead of pure packet retransmission. The protocol embeds multiplexing via *objects*. A NORM source estimates its sending rate using a variation of the TFMCC specification [RFC4654]. NORM receivers estimate their own view of the loss rate, then estimate their reception bit rate using Equation (1.8). Receivers advertise their rate in response to NORM_CMD(CC) queries emitted by the source. The source sets its sending rate to the slowest advertised value. To avoid message implosion, the source echoes the slowest advertised sending rate on the multicast channel. Receivers with a higher rate thus avoid sending their own rate.

1.8.4 Multicast deployment issues

The potential benefits of multicast raised the research community's interest in the 1990s, arguably the golden age of multicast protocols. The combination of several reasons prevented its widespread inter-domain deploy-

ment. Some technical causes have been identified and discussed in the literature [Dio+00]. We review some of them, as they lay the foundation of this thesis, but acknowledge that multiple other reasons, e.g., political and financial, contributed to the lack of Internet deployment of multicast.

Multicast state on routers. PIM requires each router on the multicast tree to maintain per-group state. However, the memory available for multicast state is often limited on routers, thus constraining the scale of possible multicast communication in large networks. There are two possible reactions when a router receives a new PIM Join, and its state is full, depending on the router's strategy. First, it drops the coming Join, thus stopping the creation of the tree for this group. Therefore, receivers will never receive the multicast traffic for this group. Second, it can broadcast the traffic linked to groups with no state due to memory saturation, which can lead to network flooding, and ultimately to numerous packet losses due to high bandwidth usage [Vig+10].

Lack of inter-domain multicast. Intra-domain multicast worked well, motivating network operators to try establishing multicast connectivity between domains (e.g., the MBONE). However, inter-domain multicast requires trusts between entities and dedicated protocols, such as multicast extensions for BGP [RFC4760]. The lack of efficient and easy-to-deploy inter-domain mechanisms constrained multicast to intra-domain boundaries. This ultimately led to the *chicken-and-egg* problem of multicast: to motivate operators to deploy inter-domain multicast, an important use-case must be developed, while application developers will not rely on the technology until it is deployed.

Receiver model mismatch and security considerations. Multicast was designed with a *receiver-driven* model, i.e., receivers choose which multicast group to listen to, without the source's consent. Worse, a source inside an IP multicast network does not know the exact set of receivers listening to the group. A misbehaving endpoint injecting packets can impact numerous receivers, as each packet is replicated in a multicast network, thus amplifying the effect of injection attacks.

The *all-or-nothing* of multicast. Due to the aforementioned issues, deploying multicast became an *all-or-nothing* problem: to receive multicast content, hosts need an end-to-end working multicast network from the source. A single router along the path between the source and the receiver without multicast support entirely breaks reachability. Similarly, multicast transport protocols, such as NORM [RFC5740], assume end-to-end multicast connectivity from the source to the receivers, without fallback mechanisms in case (subsets of) receivers lie in unicast-only networks, further limiting the attractiveness of multicast for application maintainers.

1.8.5 Application-layer multicast

The lack of inter-domain multicast deployment led researchers to develop protocols to construct multicast trees at the application layer, using unicast protocols beneath. End-System Multicast [CRZ00] first constructs an overlay network directly from the receivers, then builds an application-layer multicast tree from it. Packets are forwarded along the tree using hop-by-hop unicast. Overcast [Jan+00] is another application-layer multicast architecture building an overlay network from the participating end-hosts. These methods enable multicast across domains at the cost of efficiency, since all traffic is replicated at the application layer, potentially sending multiple copies of the same packet over the same physical link.

To bridge native and application-layer multicast, Transparent Hybrid multicast [RFC7046] proposed a common multicast API that allows both underlay (native IP multicast) and overlay implementations, allowing applications to transparently use the most efficient multicast service available in the network. The API also supports building gateways that interconnect separate multicast domains across the Internet.

1.8.6 Recent Advancements in Multicast

Recent protocols have been proposed and standardized to address some of the issues stemming from the initial design of multicast.

The Bit Index Explicit Replication (BIER) architecture [RFC8279] solves the scalability problem stemming from per-group multicast state by embedding the multicast tree *inside* the packet header. Instead of explicitly building the trees, a BIER ingress router encapsulates packets with a BIER header that contains a *bitstring*, where each bit identifies an egress router in the network. A bit with a value of 1 indicates that the corresponding egress node must receive the packet. BIER leverages the underlying routing algorithm to reach all the intended egress routers, while intermediate routers use simple bitwise operations to forward packets without creating forwarding loops. Only the ingress router must keep per-group state, i.e., the bitstring corresponding to a (S, G) tuple, since the other routers rely only on the bitstring to forward the packet along the implicit multicast tree. The drawback of this approach is that the bitstring consumes space in the data-plane, proportionally to the size of the network. BIER-TE is an extension allowing custom traffic-engineering techniques, e.g., to bypass the shortest-path routing algorithm [RFC9262].

Automatic Multicast Tunneling (AMT) [RFC7450] addresses the inter-domain deployment problem by allowing receivers in unicast-only networks to still benefit from multicast delivery without network support, by creating UDP-based tunnels. AMT introduces two components: the *AMT Gateway* running on the client, and the *AMT Relay*, a node at the edge of a multicast-

capable network that serves as the tunnel's ingress. When the client wants to receive multicast traffic, its gateway establishes a tunnel to the relay, following the AMT handshake procedure [RFC7450]. The relay then joins the tree in its multicast-capable network, using the standard IGMP/MLD procedure. Then, all multicast traffic is encapsulated in the established UDP tunnel to the AMT Gateway, finally reaching the client's application. This process is transparent from the receiver application's perspective since the decapsulated traffic is multicast, even though the underlying network does not support multicast routing. AMT directly addresses the *all-or-nothing* problem by enabling incremental multicast deployment: multicast-capable networks benefit from multicast efficiency, while unicast-only clients receive traffic through tunnels. AMT Gateways discover AMT Relays through anycast and DNS extension [RFC8777].

Tree-DN, standing for Tree-based Delivery Network [RFC9706], is the first application that leverages AMT to distribute video streams using multicast over the Internet. In practice, sources send content over GEANT, the European research network that supports IP Multicast and hosts AMT Relays. As a concrete deployment example, the VLC media player was extended with an embedded AMT Gateway daemon, enabling end-users to receive multicast streams from GEANT's AMT Relays without any network-level multicast support on their side. EUMETCast [EUMETCAST], EUMETSAT's distribution system for meteorological data, is another example of application using AMT in GEANT.

A High-Speed Robust Tunnel using Forward Erasure Correction

2

In this chapter, we leverage spare bandwidth to reduce application latency during loss events by transparently protecting traffic with Forward Erasure Correction at the network layer. We design, implement, and evaluate a high-speed extension of IPv6 Segment Routing (SRv6) in that perspective. This SRv6 extension aims to provide a tunnel in which packet losses can be recovered on time to improve application latency, transparently for the applications that do not need to implement FEC themselves.

Because this tunnel operates at the network layer, it must process packets at high speed. We thus focus on an efficient design and software implementation, capable of handling >50 Gbps of traffic under heavy losses. When run over Starlink [STARLINK], a real network that exposes non-congestion-induced losses [Mic+22b], we further demonstrate the benefits of transparently protecting data-plane traffic to improve the tail latency of HTTP responses and file system operations. This chapter led to a publication at the International Conference on Network Protocols (ICNP) in 2024 [NMB24].

2.1 Introduction

While the Internet used to be limited by the available bandwidth of a flow, this over-provisioned network capacity shifts the priority to improving the latency of modern applications. As such, content providers seek optimizations to reduce the time to completion of HTTP requests [Tre+21], especially focusing on the latency of the slowest responses – referred to as tail latency. Studies have shown that accepting a slower 1 % response time for web services can seriously erode market share [Jak; Aka]. For instance, this requirement led to 5G networks being built with low latency as a key feature [SYJ19; Sac+18b; Sac+18a]. Other modern use-cases require low latency, including automotive, augmented reality, online gaming, and future applications such as telemedicine. In these contexts, a slow response can result in poor user experience at best or serious casualties at worst.

Standard Selective-Repeat ARQ (SR-ARQ) loss recovery mechanisms used

by standard transport protocols, such as TCP [RFC9293] and QUIC [RFC9000], are not well-designed to provide low latency in lossy networks. First, time-consuming retransmissions can take *several* round-trip times [RFC2988]. In case of high end-to-end delays, the added retransmission time can severely impact the application and, consequently, the quality of experience. Second, retransmission mechanisms require resources on the sender and the receiver. The sender must manage retransmission timers, and both endpoints must maintain buffers to store out-of-order data or to retransmit potential losses. This can be problematic for devices with limited memory and computing power, such as embedded systems. Long flow completion times also drain battery life on portable devices by keeping network chips awake [Wel22].

Finally, on networks that expose *non-congestion*-induced losses, such as Starlink [Mic+22b] and 5G Fixed Wireless Access (FWA), loss-based congestion control algorithms may inadequately *decrease* the transmission rate at the source. Network service providers know that packet losses can negatively affect the performance their customers perceive. To cope with packet losses in access networks, some network operators have deployed middleboxes that intercept TCP connections to improve their performance [Xu+15]. However, with the growth of QUIC [RFC9000] and the use of encrypted tunnels for VPNs, network operators cannot deploy middleboxes that intercept all these connections. Recent works, such as Sidekick, show that encrypted data can still provide benefits when using such middleboxes for QUIC traffic [Yua+24].

Forward Erasure Correction (FEC) can be used to alleviate the cost of retransmissions [Bad+17; MDB19; Coh+20b; Kim+12; Det+15; Mic+22a]. It adds redundancy packets *a priori*, in contrast to ARQ mechanisms [Wel82], which act *a posteriori*, retransmitting packets after receiving feedback from the receiver. The added delay is significantly reduced as it corresponds to the arrival of the redundancy packets, which are independent of the RTT. This delay benefit comes at the cost of increased memory, CPU, and network bandwidth use compared to classical SR-ARQ mechanisms [COM10]. Indeed, generating the repair symbols on the encoder requires numerous linear combinations of bytes with the Random Linear Code (RLC) encoding scheme, while recovering (one or more) symbols requires solving linear systems of equations on the decoder. If not carefully designed, this can slow packet processing, ultimately leading to decreased performance. FEC consumes more bandwidth than SR-ARQ, since the encoder generates the repair symbols *a priori*. Even if adaptive coding algorithms, such as the one we present in this chapter, dynamically adjust the redundancy rate based on the perceived loss rate, the encoder usually sends more repair symbols than an SR-ARQ mechanism, which retransmits only lost packets (excluding spurious retransmissions). In light of the bandwidth over-provisioning of network operators, we argue that this

additional bandwidth consumption, though it should be minimized, is negligible in most situations. When bandwidth is limited, it should be possible to disable FEC protection. We also concede that FEC requires additional resources for mathematical computation, with potential energy considerations that we do not discuss in this chapter.

We leverage the fact that many network operators are replacing their core IP or MPLS network with Segment Routing [Fil+15]. In particular, the deployment of IPv6 Segment Routing [RFC8754; Tia+24] provides support for network programming [RFC8986], enabling operators to deploy advanced services in their networks. This chapter proposes the High-Speed Robust Tunnel (HIRT), which implements FEC at the network layer using IPv6 Segment Routing. As further detailed in this chapter, HIRT enables the protection of multiple segments simultaneously or even an entire backbone inside a tunnel formed by intermediate nodes in the network such as routers. FEC can be suggested as a service to end-hosts wishing to optimize the latency of their applications without requiring them to implement FEC themselves. Multiple end-hosts may benefit from the same tunnel simultaneously without compromising the security and integrity of their communication. Tunnels play an important role in the Internet today. They are used for Virtual Private Networks [Ber06], networks using orthogonal technologies [BWL07; Sch21; PB20], MPLS [Arm00], Segment Routing [Fil+15] and even in cellular networks [Pra+16].

Three main questions arise when considering FEC at the network layer, which we discuss in the following sections:

1. What are the benefits of adding FEC in SRv6? Should not FEC be implemented in the transport or application? (Section 2.3);

2. How to adapt the ratio of redundancy packets to the network conditions to avoid wasting too much bandwidth while offering good enough protection to recover losses in the tunnel? (Section 2.5);

3. How to efficiently generate redundancy packets and provide fast lost data recovery to simultaneously protect distinct flows? (Section 2.6).

The use of FEC below the transport layer to protect aggregated traffic across data centers has already been explored by Maelstrom [Bal+11]. Maelstrom uses multiple interleaved XOR coding windows to recover from burst losses.

We provide an implementation of HIRT leveraging IPv6 Segment Routing (SRv6) [RFC8754] and its network programmability [RFC8986] to deploy FEC tunnels protecting specific segments in the network. Figure 2.1 shows an example of the overall architecture of HIRT to protect a VPN session. The *Customer Edge* (CE) encapsulates the traffic from the *Client* with an SRv6 header

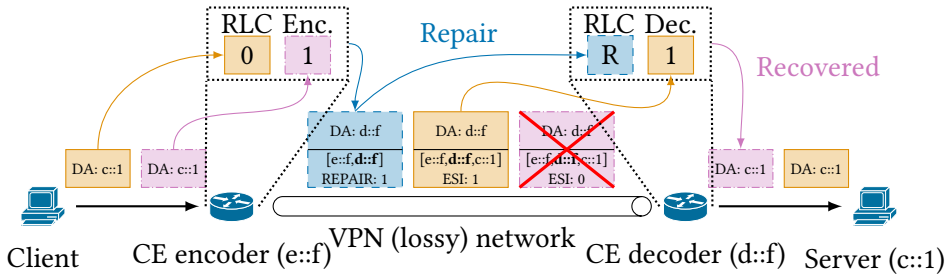


Figure 2.1: Architecture of HIRT to protect VPN traffic. The PE encapsulates the packets from its client with an SRv6 header and adds the *Encoder* and *Decoder* SIDs to protect the packets with HIRT. The *Encoder* also assigns an Encoding Symbol ID (ESI) to each source symbol and generates the repair symbols (dashed packet) using the RLC coding scheme. The CE decoder recovers the losses using the received symbols (source symbol with ESI=1 and the repair symbol) and decapsulates the SRv6 header of the source symbols before transmitting the packet to the server. The repair symbol does not exit the tunnel.

to trigger FEC protection without requiring the Client or the Server to implement FEC themselves. HIRT leverages the Convolutional Random Linear Code (RLC) [RFC8681] to dynamically adapt the amount of redundancy. We implement HIRT in FastClick [BSM15] with several optimizations, including kernel bypass and a scalable multithreading architecture. We show that HIRT can protect >50 Gbps under >3 % losses.

We further demonstrate the effectiveness of HIRT over the Low Earth Orbit (LEO) satellite network Starlink [STARLINK], which exhibits *non-congestion* induced losses [Mic+22b] that can double the flow completion time of short requests. We use HTTP benchmarks to show that HIRT can effectively recover lost packets independently of the transport protocol and reduce this tail latency by up to a factor of 2. HIRT also reduces the request completion time of longer responses by several RTTs by recovering packet losses without impacting the end-host's transport congestion controller. Finally, we highlight the benefits of HIRT on a network file system benchmark using Starlink. We show that the tail latency is reduced by up to 20 %. The source code of Maelstrom is not publicly available for comparison. As such, it is not possible to compare the performance in high-speed environments. As an indication, in 2011, Maelstrom could protect up to 700 Mbps of traffic. Instead, we build an *ad hoc* packet simulator generating reproducible losses and implement HIRT and Maelstrom to further compare their performance in Section 2.9.2.

Contributions. We make the following contributions in this chapter:

- The design of HIRT inside IPv6 Segment Routing.
- An adaptive algorithm to schedule the generation of repair symbols based on loss estimation feedback and RLC.
- A high-speed implementation of HIRT, enabling >50 Gbps of traffic protection under heavy losses.
- Evaluations of HIRT in a real network exposing transmission errors (Starlink) under HTTP and file transfer benchmarks.
- A packet simulator where we compare HIRT and the closest state-of-the-art, Maelstrom [Bal+11] under reproducible loss patterns.

The remainder of this chapter is organized as follows. We first provide a reminder about IPv6 Segment Routing in Section 2.2. Then, we highlight the benefits of FEC at the network layer compared to other approaches in Section 2.3. We detail our design of HIRT at the network layer with IPv6 Segment Routing in Section 2.4. Section 2.5 presents our adaptive FEC scheme to dynamically respond to the estimated network conditions. Section 2.6 presents our approach and the challenges to enable high-speed FEC on routers. We benchmark the system in Section 2.7 and evaluate its impact over Starlink in Section 2.8. Section 2.9.2 compares HIRT to Maelstrom through simulations. Finally, we present the related works in Section 2.10 and conclude this chapter in Section 2.11.

2.2 IPv6 Segment Routing

Routing protocols populate the forwarding tables of routers in a network. Through what we commonly call the *control plane*, routers exchange topological information, ultimately letting them have a global view of the network. To determine the path packets follow, routing algorithms usually compute the *shortest path* towards all other nodes in the network. Based on the IP destination address, routers then use this shortest path to forward packets.

However, this mechanism only uses the *shortest-path* policy based on shared topological information, whereas use-cases may require other, specific paths to send packets depending on specific source/destination pairs. For example, network operators may want to use longer, less-optimized paths for backup applications to keep available bandwidth on the faster path for sensitive applications. This class of routing strategy is called *traffic engineering*, and led to the development of *source routing*. In source routing, the source partially or entirely defines the path a packet takes to reach the destination, bypassing the network's underlying routing algorithm.

IPv6 Segment Routing (SRv6) [RFC8754; RFC8402] enforces the specific behavior to execute *inside* the packet. An SRv6 source defines a list of

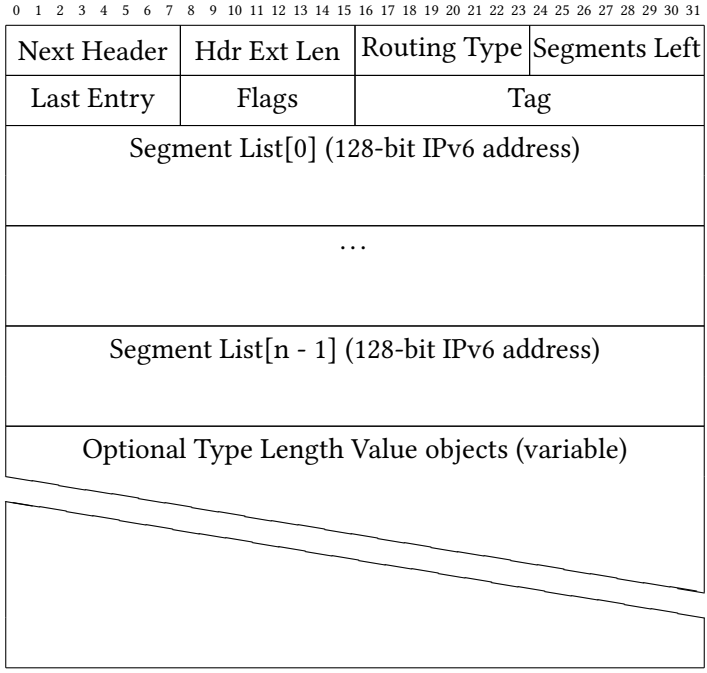


Figure 2.2: SRv6 Segment Routing Header (SRH) format.

segments, indicating the intermediate endpoints a packet must traverse, and *how* these intermediate nodes must forward it. Only the segment endpoint processes the specific forwarding method defined in the segment; all non-endpoint routers forward the packet using the underlying routing algorithm, usually the shortest-path algorithm. IPv6 Segment Routing is implemented in the Linux kernel version 4.10 [LB17].

2.2.1 IPv6 Segment Routing Header

IPv6 Segment Routing (SRv6) [RFC8754] uses the IPv6 data plane in the segment routing architecture. Segments Identifiers (SIDs) are represented as IPv6 addresses, i.e., each segment is 128 bits long, and the *segment list* consists of a list of the IPv6 addresses of intermediate endpoints. The ingress router of the SRv6 data plane adds an SRv6 header, which is represented in Figure 2.2. The router either adds the header after the existing IPv6 header or completely encapsulates the packet with new IPv6+SRv6 headers. The segment list works as a reversed stack: the first segment to process is the last segment (index $n - 1$), all the way back to the top. The *Segments Left* indicates the index of the active segment in the list and is decremented at each endpoint, while the *Last Entry* indicates the index of the last segment in the list. Like other

IPv6 routing headers [RFC8200], SRv6 defines *Type-Length-Value* (TLV) options. The *Type* and *Length* are 8 bits long. The *Length* indicates the length of the *Value* field. SRv6's specification [RFC8754] defines *Padding* TLVs, as the SRv6 header must be a multiple of 8 bytes long.

2.2.2 SRv6 Packet processing

Algorithm 1: SRv6 packet processing.

```
1 if IPv6 Destination Address ∉ local SID table then
2 |   Forward packet using standard IPv6 routing          ▷ Transit node
3 else
4 |   if Segments Left = 0 then
5 |   |   Remove SRH and deliver to next header          ▷ Final endpoint
6 |   else
7 |   |   Segments Left ← Segments Left −1
8 |   |   IPv6 Destination Address ← Segment List[Segments Left]
9 |   |   Execute network programming function of the previous
  |   |     (finished) segment
```

Algorithm 1 summarizes SRv6 packet processing on a segment routing-aware router. For clarity, we omit parts of the algorithm related to IPv6 Hop Limit verification. When a packet is SRv6-encapsulated, we can distinguish *endpoint* routers from *transit* routers. An endpoint router is the termination point of a segment; i.e., one of the segments in the list corresponds to a local segment identifier of this router. There are as many endpoints as there are segments in the list. A transit router is not a segment termination, and simply forwards the packet to the active segment endpoint. The concept of segmenting the path is reflected in the IPv6 header: the IPv6 header's destination address is set to the segment identifier of the active SRv6 segment, allowing a transit router to forward the packet *solely* based on the IPv6 destination address (Line 2).

A segment termination (i.e., endpoint) processes the packet differently (Lines 4-9). If it is the last segment of the list (*Segments Left* = 0), it removes the SRv6 header and processes the next header in the IPv6 headers chain, potentially processing subsequent protocols, e.g., TCP (Line 5). Otherwise, because the router is the endpoint of the current segment, it updates the packet's destination address to the next segment in the list – by decreasing the *Segments Left*. Line 9 indicates local processing of the segment by the endpoint, which relates to the *Network Programming* paradigm [RFC8986].

2.2.3 SRv6 Network Programming

The SRv6 Network Programming framework [RFC8986] allows operators to define specific functions to execute on segment endpoints. Line 9 of Algorithm 1 refers to the execution of a local function associated with the active SID, which is the core of the SRv6 Network Programming framework. In practice, these functions are encoded in the segment identifier (SID), which is an IPv6 address. The framework leverages the large 128-bit long address space to semantically split an address into *LOC*:*FUNC*:*ARGS*, ensuring that $|LOC| + |FUNC| + |ARGS| \leq 128$:

- *LOC* stands for *locator*. In practice, this is (one of) the endpoint's prefixes, allowing transit routers to forward the packet to the correct segment's termination point;
- *FUNC* refers to a specific function that the endpoint applies before/when processing the packet;
- The *ARGS* embeds parameters to further specify the behavior of the function to execute on the packet.

The exact mapping between *FUNC*/*ARGS* and the associated functions/arguments is left to the network operators to define, but once the policy is determined, this framework allows SRv6 ingress routers to define specific packet processing along a determined path to reach the destination.

Multiple SRv6 network programming functions have been standardized to allow strong traffic engineering capabilities [RFC8986; Aub+16; Jad+19]. For example, the End function defines the packet's basic processing: after decrementing the *Segments Left* and updating the IPv6 Destination Address, the endpoint forwards the packet according to the underlying routing algorithm's policy. The End.X function asks the endpoint to forward the packet at a specific outgoing interface, even if the underlying routing algorithm would use another interface to reach the new segment's locator.

While this set of functions [RFC8986] is statically defined, operators can theoretically define other functions. For example, the End.BPF functions have been studied in the literature [XDB18], proposing the dynamic creation of new functions using the eBPF virtual machine within the Linux kernel, an extension of the BSD packet filter [MJ+93].

2.3 Network Layer Forward Erasure Correction

FEC techniques are typically applied at the datalink [CZT99; ZZ11; Fuk+10] and the transport layer [MDB19; SMR18; RSM18; LK04; COM10; De +19; Coh+20a; RV98; TZ01; Lub+02], respectively, to protect a single link or an

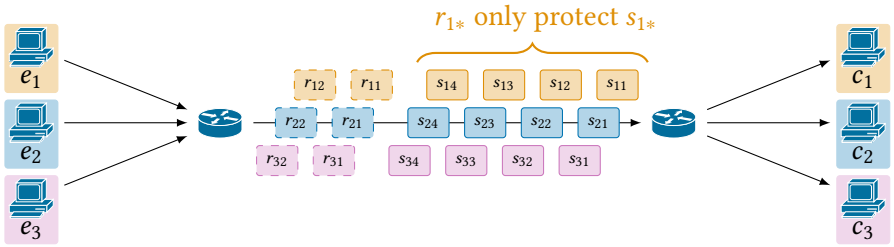


Figure 2.3: Transport-layer FEC. Each end-host generates its own repair symbols (r) to protect its source symbols (s) individually.

entire communication path between two end-hosts. At the transport layer, the FEC mechanism can leverage knowledge from the application [Mic+22a; MB24], e.g., to protect only specific application objects. While this may seem counter-intuitive at first glance, end-to-end FEC also enables faster recovery from congestion-induced losses than retransmissions in particular scenarios [Mic23].

Despite losing this application-level knowledge with network-layer FEC, we offer benefits over transport-layer FEC. First, the computation is performed on intermediate routers, enabling end-hosts to leverage the mechanism without implementing it themselves. This is particularly interesting when we consider the heterogeneous landscape of implementations for the same protocol (e.g., more than twenty QUIC stacks coexist in the wild [IET25]). Extending the protocol with FEC requires standardization, which might take several years, and even more time to be deployed [Wir+20]. This FEC offload comes at the cost of more processing on the encoding and decoding routers. We analyze this overhead in Section 2.7.

Second, network-layer FEC enables aggregation of traffic from multiple sources to increase the overall flow from the encoder to the decoder. This is illustrated in Figure 2.3 and Figure 2.4, respectively, for transport-layer and network-layer FEC. In the first case, each sender e generates a set of repair symbols r to protect its own traffic. In the Figure, every sender must generate two repair symbols to be able to recover any loss of two packets. This strategy results in a total of six additional packets entering the network.

With network-layer FEC, the encoder and decoder nodes aggregate the three flows to generate fewer repair symbols (see Figure 2.4). Aggregating numerous flows helps conserve resources, as the same redundant packets can protect multiple flows. When losses occur in bursts, this prevents every transport sender from sending numerous redundant packets for each flow, allowing long bursts to be recovered. For example, r_1 and r_2 alone can recover any pair of erased source symbols.

Third, at the transport layer, the FEC mechanism cannot make any as-

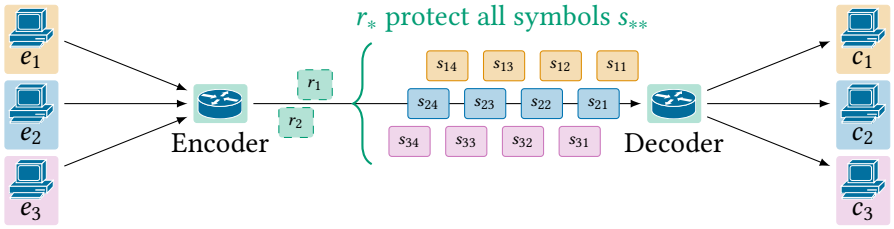


Figure 2.4: Network-layer FEC. The encoder aggregates the three flows, generating only two repair symbols (r) that protect the source symbols (s) of all flows simultaneously.

sumption about the paths its packets follow. At the network layer, one can dynamically deploy FEC to protect specific links or paths that exhibit *non-congestion* induced losses. It is also possible to deploy multiple independent instances within a given network. The flexible deployment of the tunnels enables network operators to make several assumptions about the link or path to protect, such as the available resources of the routers (buffer sizes, network interface controllers, link reliability, etc.). When the network is over-provisioned for bandwidth, one may assume that packet losses are mainly due to transmission errors. When the error is due to congestion within the tunnel, the decoder can mark the recovered packet with ECN [RFC3168] to notify the end-user stacks. The encoder can also mark repair packets with a lower quality-of-service level so that routers along the tunnel drop them first.

When operating over a link with non-congestion-induced packet losses, recovering these packets at the network layer and hiding those loss events from the transport layer avoids an unnecessary decrease of the transport layer congestion window. Hiding loss events to the congestion controller is explicitly discouraged at the transport layer, where the origin of loss events is undetermined [RFC9265]. By deploying HIRT on links where losses are known not to be congestion-induced, erasures can be recovered and hidden from the transport layer to increase the throughput of transport connections with no harm to the network. We highlight this behaviour in Section 2.8.1.

2.4 Integration in IPv6 Segment Routing

This section presents the design of HIRT within IPv6 Segment Routing (SRv6) using the Network Programming paradigm [RFC8986]. Similar approaches can be developed to integrate network-layer FEC in other protocols, e.g., MPLS [RFC3031]. Operators often deploy SRv6 for intra-domain purposes, but its deployment is increasing on the Internet [SR]. As a result, HIRT is dependent on the deployment of SRv6.

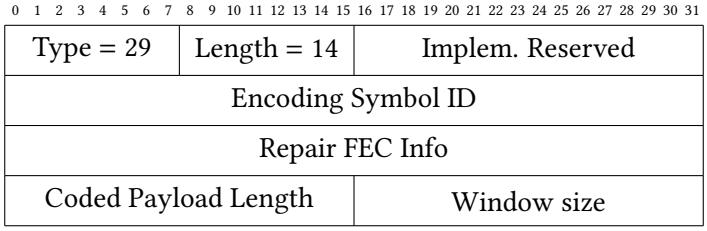


Figure 2.5: HIRT SRv6 repair symbol TLV. The source symbol TLV uses a *Type* of 28, *Length* of 6 and stops after the *Encoding Symbol ID* field, totaling 8 bytes.

2.4.1 HIRT in SRv6 Network Programming

We leverage the *Segment List* in the IPv6 Segment Routing header to deploy FEC on routers. The two SIDs of the encoder and decoder are carried in the *Segment List* of the SRv6 header. This SRv6 header can be added by the encoder, an upstream router, or the source (in Figure 2.1, the CE adds it). In the first case, the network operator may decide to protect all traffic to a given destination without requiring applications to be aware of the tunnel. In the second situation, any host may decide to use the tunnel for its specific traffic. Only applications that may benefit from HIRT (e.g., latency-sensitive ones) can activate it by adding the SRv6 header. This list is also used to recover some fields of the IPv6 packet header that may be altered by intermediate routers within the tunnel, such as the destination address, which differs when a packet is processed by the encoder and decoder routers.

2.4.2 Source and repair symbols representation

HIRT protects data at the packet level. Thus, the source symbols are IPv6 packets exchanged between end-hosts via the tunnel. Using packet-level protection enables the tunnel to make *no assumptions* about the data it protects, and it does not require the encoder and the decoder to analyze the content of the packets. HIRT processes SRv6 packets as payload without compromising the security of communication.

Each source symbol is uniquely identified by an *Encoding Symbol ID* (ESI), a 32-bit value incremented by one for each source symbol protected by the encoder. To carry the ESI, the encoder adds an SRv6 *Type-Length-Value* (TLV) option [RFC8754]. Routers that do not recognize the TLV type skip it during the packet processing since the highest-order 2 bits of the *Type* are set to 0 [RFC8200]. This allows HIRT to be forwarded by existing routers that do not support our system.

The repair symbols carry the *Repair FEC payload*, i.e., the redundancy data used to recover lost packets. They are generated by the encoder and

sent to the decoder (Figure 2.1). They are not forwarded outside the tunnel. The repair payload is encapsulated in an SRv6 header to be routed to the decoder. This forms the repair packet. A repair symbol is also uniquely identified, using the ESI of the last source symbol it protects in its window. The representation of the repair symbol TLV is given in Figure 2.5. It also contains FEC scheme-specific information, e.g., the RLC seed used to generate the pseudo-random coefficients of the repair symbol (*Repair FEC Info*), as well as the number of source symbols protected by the current repair symbol (*Window size*). Because packets can be of different sizes, the encoding and decoding processes must first *pad* them by adding zeros at the end of the packet to reach 1500 B (we assume that the MTU is 1500 B). To retrieve the original length (i.e., remove the padding) of a recovered symbol, the *Coded Payload Length* contains the FEC-encoded packet size, using the same algorithm as the Repair FEC payload. Finally, the SRv6 TLV of repair packets contains a field indicating which FEC algorithm was used to generate the repair symbols. HIRT only implements RLC, but network operators could implement, e.g., XOR-based algorithms. Both the source and repair TLV additionally contain the *Implem. Reserved* 16-bit field. As indicated by the name, the meaning of this field is left to the implementation. It ensures that the TLV is a multiple of 8 bytes [RFC8200]. We leverage this field in Section 2.6.

2.5 Adaptive FEC

The use of Forward Erasure Correction involves a trade-off between transmission delay and bandwidth consumption [Hui96]. Adding redundancy increases the probability of recovering lost data, reducing the transmission delay, but it also increases the bandwidth usage. Several works suggest schemes to adapt the coding rate of FEC based on feedback [BFT99; AHH05; Coh+20a; Coh+20b]. However, these designs focus on providing (partial or full) reliability at the cost of added delay and do not prioritize efficient packet processing. We design an adaptive algorithm to schedule repair symbols with three considerations: (i) it must perform well in heterogeneous networks, such as those with varying delay or reliability (e.g., satellite networks); (ii) the resources required to generate the feedback must remain adequate to ensure a high-speed delivery; and (iii) no assumptions can be made about the protected traffic. For example, it may be a bulk download, a video conference call or a mix of both.

Our system leverages feedback from the decoder to the encoder to estimate the loss events in the tunnel. The amount of generated repair symbols is adjusted using this estimation. HIRT leverages the sliding-window RLC algorithm [RFC8681] and adapts the step size according to the adaptive algorithm described in this section.

2.5.1 Loss estimation and feedback

Parameter	Description
β	Factor to take into account the standard deviation in loss estimation, $\hat{\sigma}$ when computing the redundancy ratio.
n_f	Number of data-plane packets before sending feedback from the decoder to the encoder with new loss estimations.
w	FEC sliding window size.

Table 2.1: Parameters introduced in HIRT.

The decoder sends feedback messages to the encoder inside the data plane. Concretely, it regularly sends an SRv6 packet with feedback information inside an SRv6 TLV option [RFC8754] of *Type* 30. These messages contain the estimated loss characteristics from the decoder's viewpoint since the last feedback. Because each source symbol is identified with a monotonically increasing Encoding Symbol ID (ESI), the decoder marks a packet as lost if it sees a gap in the ESI sequence. We discuss packet reordering in Section 2.6. Table 2.1 summarizes the parameters introduced in this chapter. Feedback is computed and forwarded every n_f packets. The decoder returns the window size n_f and the number of missing ESI in that window. Other strategies to send this feedback exist, e.g., using a TCP connection between the encoder and the decoder to send this feedback reliably. Instead, HIRT uses the SRv6 data plane as the implementation is optimized to process these packets efficiently. Similarly to the repair packets, these feedback SRv6 packets do not exit the tunnel and the encoder drops them after processing. The *SRv6 Segment List* contains a single entry for the encoder's address (`e::f` in Figure 2.1).

With this feedback, the encoder adjusts its estimation of the mean loss inside the tunnel, $\hat{\mu}$. The algorithm uses a moving average update to give more importance to recent estimations to adjust to the heterogeneity of the network. The encoder also computes the standard deviation in the mean number of losses, $\hat{\sigma}$. This feedback is easy to compute both for the encoder and the decoder: the decoder must only keep a bitmap of size n_f to store the state of received ESI. The loss of feedback messages does not prevent the encoder from producing FEC repair packets; only its view of the network condition is potentially impacted until the next feedback is received. Finally, no assumptions are made on the protected traffic, except that we analyze the network condition based only on protected packets.

2.5.2 Adaptive algorithm

The encoder generates repair symbols to compensate the estimated loss distribution using the feedback mechanism. As the number of repair packets to generate is based on the previous estimation (and thus computed *a priori*), the encoder may generate too many or too few symbols. The first case results in wasted bandwidth; the latter in the impossibility of recovering all lost symbols. To compensate for the $\hat{\mu}$ estimated symbols that will be dropped within the next n_f sent source symbols, the encoder generates at least $\lceil \hat{\mu} \times n_f \rceil$ repair symbols every n_f source symbols. The variance $\hat{\sigma}$ is taken into account in this computation. We introduce a new parameter, β, which amplifies the impact of the variance in the redundancy, such that the encoder generates $\lceil (\hat{\mu} + \beta\hat{\sigma}) \times n_f \rceil$ repair symbols every n_f source symbols. We evaluate the impact of this parameter in Section 2.7.

Two strategies exist to schedule the generation of repair packets. First, they can either be generated all at once, every n_f source symbols (*block window*). In that case, if one of the first source symbols of the window is lost, the decoder must wait for the reception of all remaining source and repair symbols to be able to recover the loss. This can potentially add a great amount of delay for the receiving end-host. The other strategy is to interleave repair symbols with the data (*convolutional window*). Evaluations of Forward Erasure Correction methods showed that the second strategy provides lower delay to recover lost source symbols without impacting the quality of recovery [Roc+17; MDB19]. HIRT relies on the second strategy using the RLC coding scheme as it can generate repair symbols interleaved with the source symbols.

The scheduler thus generates n_f repair packets every $\lceil \frac{n_f}{\hat{\mu}+\beta\hat{\sigma}} \rceil$ source symbols. *HIRT interleaves these repair packets and generates a repair symbol with a step of* $\lceil \frac{1}{\hat{\mu}+\beta\hat{\sigma}} \rceil$ *source symbols.* The step is dynamically adjusted with the feedback from the decoder.

2.6 High performance software implementation

A natural approach for network-layer Forward Erasure Correction with high-speed packet processing and recovery is to implement it in hardware directly, e.g., in P4 [Bos+14]. However, RLC requires computing linear combinations of packets (at the encoder) and solving linear systems of equations (at the decoder), which is complex and tedious to be efficiently implemented in hardware. Simpler algorithms could be implemented, such as the XOR erasure code, but its efficiency has been demonstrated to be low [MDB19] and we do not consider it in this work. Maelstrom [Bal+11] explores multi-layered XOR codes, which can protect burst losses at the cost of constant and significant

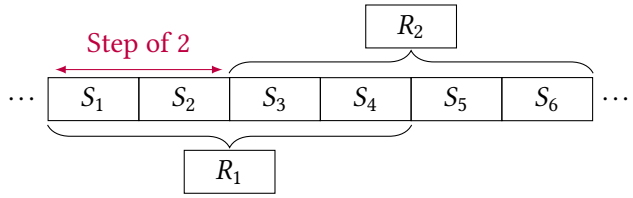


Figure 2.6: Sliding window coding with $w = 4$ (window size) and $\hat{\mu} = 0.5$ (mean loss estimation) resulting in a step of 2 between two repair symbols.

overhead. We compare Maelstrom and HIRT in Section 2.9.2.

HIRT leverages FastClick [BSM15], an enhanced version of the Click modular router toolkit [Koh+00] to build software network functions supporting traffic above 100 Gbps. FastClick integrates kernel bypass I/O such as DPDK [Fou15] and packet batching. With FastClick, one defines packet processing as a graph of smaller components that run in user-space. We extended the toolkit to support basic SRv6 processing. This required ~330 lines of code (LoC). FEC must be done carefully as it can be CPU-intensive to compute the repair symbols and to solve the RLC linear system. Packets that are not protected by the tunnel (i.e., packets without the SRv6 header with the encoder and decoder SIDs) are processed and forwarded without entering the HIRT component in FastClick. The whole implementation required ~2300 LoC.

FEC sliding window. Conventional FEC approaches use a constant and limited FEC window size, i.e., all repair symbols protect the same number of source symbols. As detailed in Section 2.5, repair symbols are interleaved with the source symbols they protect. HIRT uses a constant sliding window of size w, and we set $n_f = w$ all the time. In steady state, a new symbol entering the window removes the oldest one. Figure 2.6 shows an example of six source symbols protected by two repair symbols and a step of 2. Repair R_1 and R_2 respectively protect sources $S_{1\to4}$ and $S_{3\to6}$. Increasing w improves recovery capabilities at the cost of more processing each time a new repair packet is generated, and a linear system is built. Carefully choosing the right value of w thus brings a trade-off between recovery capability and processing performance.

DPDK. Our implementation uses DPDK [Fou15]. The toolkit provides kernel bypass to speed up the packet processing in user-space [BSM15]. DPDK reserves beforehand a pool of buffers for the packets, thus cancelling the cost of memory allocation. Our solution also uses batching of 32 packets. These two features are already supported in FastClick. Packets shorter than the 1500 B MTU are padded using the spare bytes from the pre-allocated buffer in the pool.

AVX instruction. RLC is computationally intensive. It uses costly Galois Field operations to work with integer numbers in our equation systems. To

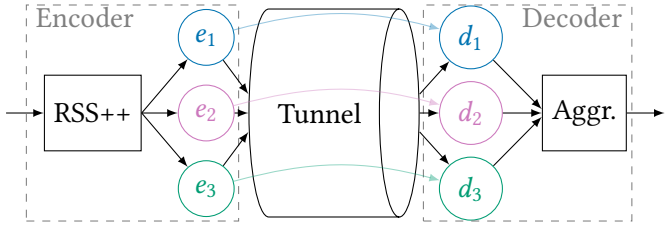


Figure 2.7: Multithreading architecture of our implementation with 3 parallel streams. The *Encoder* distributes the incoming packets among the encoder streams e_1, e_2 and e_3. Each encoder stream e_i communicates with its decoder stream d_i. All packets are independently sent with the same tunnel but each stream is independent from each other. At the end of the decoding streams (d), the traffic is aggregated and forwarded. The curved arrows between the e_is and d_is represent the streams.

alleviate the cost, we use the `moepgf` C library [MOEPGF] which uses AVX SIMD instructions.

Zero copy. The source symbols are kept in memory to generate the repair symbols (at the encoder) and construct the linear system to recover lost packets (at the decoder). Instead of copying the packet in separate buffers, our implementation keeps a pointer to the packet buffer and updates the DPDK buffer's reference counter. Memory is only released once every reference to the same packet is no longer valid, i.e., the packet has been processed and is removed from the FEC buffers. The packet may be modified while being kept by reference counting. It happens when the SRv6 header is modified after the encoding or decoding operation is applied, e.g., to change the IPv6 Destination Header field and decrement the Hop Limit. FastClick prevents concurrent writing by automatically copying shared packets when an attempt to get a writable reference to a shared packet is made. This adds unnecessary copies that reduce the performance. Instead, the implementation only keeps a copy of the fields that might be modified and manually fixes the modified buffer upon repair symbol generation and system solving. The size of these fields does not exceed 40 bytes. This trick allows disabling the copy-on-write security of FastClick and keeping only one copy for each packet.

Multithread architecture. Our implementation supports multithreading. The bottleneck operations are the generation of the repair symbols, and the construction and the solving of the linear system. When a CPU core performs one of these operations, it cannot process other incoming packets in its queue. Our implementation scales by using several cores in parallel. For this, we split the traffic into parallel streams.

Figure 2.7 presents the multithread architecture with 3 concurrent processing threads. At the encoder, the network interface controller (NIC) distributes incoming packets among the available CPU cores. This distribution

uses RSS++[Bar+19] to spread the load equitably among the cores and dynamically scale according to the input load. RSS++ maps packets from each flow to the same CPU core to avoid possible reordering.

There is a one-to-one mapping between an encoding thread and an available CPU core. A *Stream ID* is assigned to each encoding thread. There is no communication between encoding threads with distinct Stream IDs. Each thread processes its packets and encodes them independently of the other threads. For example, each stream uses its own Encoding Symbol ID (ESI) space. This architecture creates several concurrent streams inside HIRT, as represented in Figure 2.7. The *Stream ID* of a source or repair symbol is carried inside the *Implem. Reserved* field of the SRv6 TLV of the source, repair and feedback packets. The decoder uses the same architecture. It uses the *Stream ID* from the TLV to map the symbol to the correct decoding CPU core. The system constructions and resolutions are also independent between streams, i.e., each stream constitutes an independent FEC tunnel from the others. The feedback messages are also stream-dependent. Each stream has its own estimation of the loss pattern inside the tunnel. This architecture enables us to arbitrarily increase the number of threads and Stream IDs *without* contention of a single performance bottleneck for the FEC computation.

Overloading detection. A vicious circle may occur in case of severe losses in the tunnel. In that scenario, the encoder decides to generate numerous repair symbols in a short amount of time, increasing its CPU load. If the load reaches the maximal capacity, the router will have to drop incoming packets, thus creating losses at the tunnel ingress. This would deteriorate the performance of the applications using the tunnel. The decoder may also become overloaded in case of severe losses as it tries to solve numerous linear systems. Similarly, it would drop packets at its ingress, thus creating more losses inside the tunnel. These drops will increase the estimated network losses. In response to this additional loss, the encoder will decide to generate more repair symbols, keeping increasing the CPU load at the decoder.

Our implementation alleviates this situation by *randomly skipping* repair symbols generation and system solving if needed. When a stream CPU core sees its load reaching a defined threshold value, it starts to randomly skip these events with a probability *proportional* to the current CPU overload. Other methods could be implemented to, for example, genuinely decide which linear system the decoder should skip instead of randomly selecting one system to discard. However, this situation occurs when the CPU load is high, and it should not require additional resources to make a decision.

Loss detection and packet reordering. The strategy to consider a packet as lost has an impact on the performance of the implementation. A missing ESI in the source symbol buffer is interpreted as a lost packet. A repair symbol is always sent by the encoder *after* the source symbols it protects, as il-

lustrated in Figure 2.1. Upon reception of the repair symbol, a missing ESI can only be due to losses or reordering. Intermediate routers in the protection tunnel can introduce reordering. A study highlighted that such events are rare but the average out-of-order length is of 8 packets [BS02]. If a source symbol is not lost and simply out-of-order, the decoder might trigger the recovery mechanism, increasing the CPU load. Two different strategies can be implemented to counter this side effect. First, upon reception of a repair symbol, the decoder can postpone the recovery mechanism for a few packets. The decoder could receive out-of-order packets and detect that a recovery is not necessary anymore. The second approach is to consider that out-of-order source symbols are already delayed for the application. In that case, recovering this source symbol means that it will be sent *faster* out of the tunnel compared to the first solution because the tunnel does not need to wait for the reception of the out-of-order packet. However, it adds CPU load because additional recoveries are triggered. Our implementation uses the second approach to decrease the impact of reordered packets, thus reducing the overall jitter for the applications using HIRT.

2.7 Benchmarks

Section 2.5 presented our adaptive FEC scheduler, introducing β, i.e., the impact of the variance of the number of lost packets in the tunnel when generating new repair symbols. In this section, we first analyze the impact of this parameter under emulated losses in a controller environment. Then, we benchmark our high-speed implementation of HIRT to identify the system's scalability limits.

2.7.1 Adaptive algorithm performance

We evaluate the performance of our adaptive algorithm compared to a more standard solution involving a fixed step size (similarly to Figure 2.6). For this, we run `iperf3` experiments between two network namespaces on an Ampere(R) Altra(R) Processor Q80-30 CPU at 2.8 GHz. An `iperf3` experiment consists of the client sending UDP 100 B-payload packets at a constant rate of 1 Mbps for 30 s. We emulate a one-way delay of 25 ms in each direction using `tc`. To evaluate our adaptive scheme, we emulate losses using a simplified Gilbert-Elliott drop model [HH08] and perform an experimental design on its parameters with 99 points [Kir12].

Gilbert-Elliott model. This model emulates burst losses by running a two-state Markov model (Figure 2.8). Packets are forwarded if the model lies in the *Good* state, and dropped if it is in the *Bad* state. The probability to go from the *Good* to the *Bad* state is p; the probability to go from the *Bad* to the

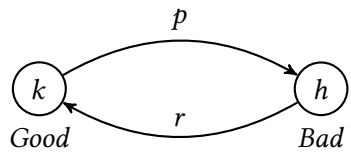


Figure 2.8: Simplified Gilbert-Elliott drop model consisting of two states. Packets are dropped in the *Bad* state.

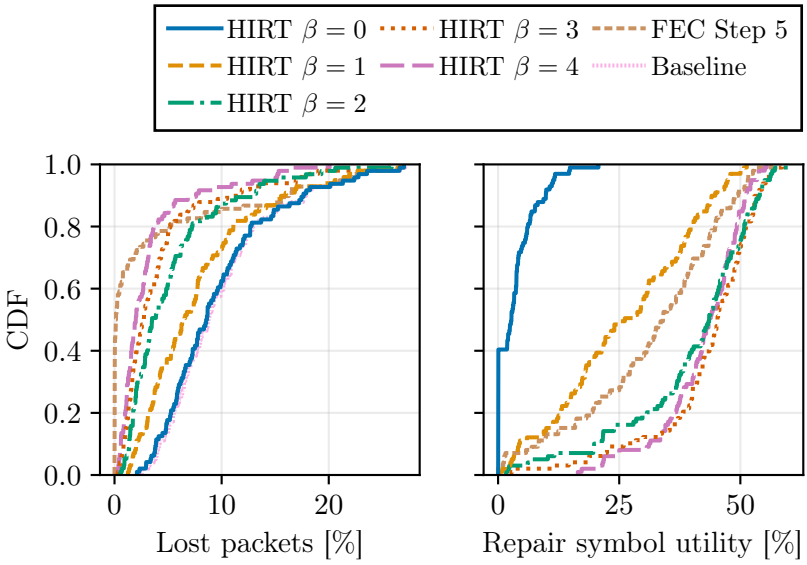


Figure 2.9: Impact of β regarding recovery capabilities under a controlled environment.

Good state is r. The more generic model also accounts for drop probability while remaining in the *Good* and *Bad* states; for simplicity, we assume that packets are erased only in the *Bad* state.

Experimental design. The experimental design approach [Kir12] removes the subjectivity in choosing model parameters. It samples points in a defined parameter space such that the distance between two points is maximized. We use the ranges $p \in [0.01, 0.1]$ and $r \in [0.1, 0.8]$ of the Gilbert-Elliott model. Such extreme scenarios do not attempt to represent reality. Instead, they show the performance and limits of our algorithm.

Figure 2.9 shows the loss percentage measured by the `iperf3` receiver. We compare the adaptive algorithm with several values of β (`HIRT-x` where β = x) with a solution considering a simple strategy of constant step size of 5 (`S-5`) and the baseline, i.e., without FEC (the `Baseline` curve). Since traffic is unreliable, this curve shows the CDF of the loss percentage. We use

a constant window size of $n_f = w = 200$ packets.

In more than half the experiments, the constant-step-size recovers all packets, since the number of repair packets (overhead of 20 %) is sufficient to recover from a small amount of losses. The adaptive algorithm never recovers all losses, even for high values of β. By analyzing the traces, we see that the loss estimation converges slowly to the correct value. During this time, the system can not recover all the losses. However, the adaptive scheme becomes more efficient for $\beta \geq 2$ in the ~20 % most extreme scenarios. As expected, β plays a major role in the number of packets we recover. Increasing its value increases the probability that the loss pattern is covered by the number of repair symbols sent. When the scheduler does not include the variance ($\beta = 0$), HIRT almost recovers none of the lost symbols. The value of β captures the variance of the estimated loss rate as losses are not evenly distributed. Generating *just enough* repair symbols ($\beta = 0$ in Figure 2.9) to compensate for the loss rate is not sufficient to recover from packet losses. By increasing β, we account for this variance in the loss rate estimation, thus generating more repair symbols, allowing for better packet recovery.

Increased recovery capabilities also come with higher packet overhead. However, not all repair symbols will be used to recover lost symbols, resulting in wasted bandwidth. We call the *repair symbol utility* the proportion of generated repair symbols that actually recovered lost packets. This is computed as the ratio of recovered packets to sent repair symbols. The results are presented on the right-hand side of Figure 2.9. Again, for $\beta = 0$, the encoder does not generate enough repair symbols to compensate for the longer burst losses. The few repair symbols thus cannot recover many source symbols.

We see that $\beta \geq 2$ produces the most useful repair symbols. More than 40 % of the generated redundancy symbols are used in 40 % of the experiments. From the Figure, we see that on average, $\beta = 3$ generates the best trade-off between loss-recovery capabilities and repair symbol utility. At higher values, the added overhead is significant for a marginal improvement in recovered packets. We use $\beta = 3$ for the remainder of the evaluations of this chapter.

2.7.2 HIRT under high-speed traffic

We assess the performance of HIRT at high speed to show that a carefully designed prototype, even when implemented in software, can support >50 Gbps of traffic under heavy losses using a few commodity CPU cores. Throughout the experiments, we set $w = 200$. We experimentally found that a higher value does not significantly improve the recovery capabilities, while the processing overhead decreases the performance at high speeds. We rely on the CloudLab infrastructure [Dup+19] to benchmark our implementation. We use a linear topology composed of five nodes, respectively the server, the encoder, a node

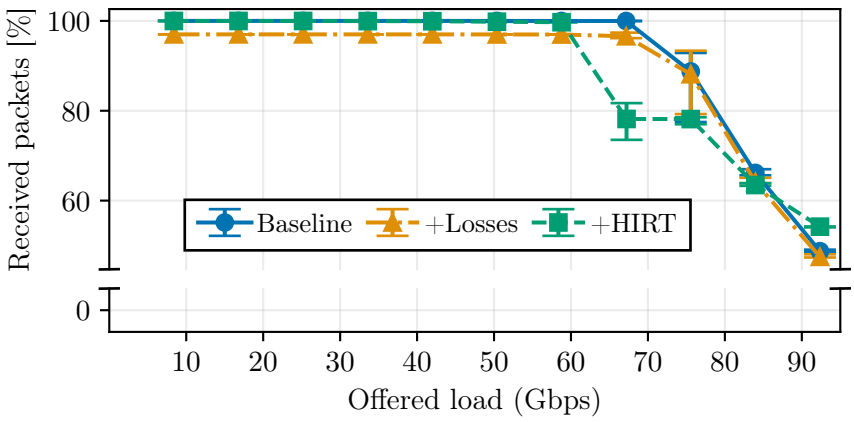


Figure 2.10: Throughput reached by HIRT under 3 % losses.

emulating losses, the decoder, and the client. All nodes are equipped with x86 AMD EPYC 7452 processors. Each node has 32 CPU cores. The L1, L2 and L3 cache layers are respectively 32 kB, 512 kB and 16 384 kB. All nodes are connected through 100 Gbps links inside the Utah cluster. We evaluate the limits of HIRT in terms of throughput.

Offered load. Figure 2.10 shows the percentage of packets received by the client given an increasing offered load, using 1024-byte UDP packets. The baseline curve (i.e., without losses or HIRT) shows that we observe losses before an offered load of 80 Gbps. This result shows that we cannot fully utilize the available 100 Gbps bandwidth and that the machines become the bottleneck of the experiment. By verifying the NIC counters, we attribute this limit to the congestion inside the NIC when using two ports on a single card.

When inducing 3 % of packets uniformly dropped, HIRT recovers all dropped packets below 60 Gbps. When the CPU load on the routers increases, the number of recovered packets is capped, as explained in Section 2.6, to avoid dropping packets due to CPU contention. Above 60 Gbps, the client receives fewer packets with HIRT than in the `Baseline+Losses` situation, i.e., coding nodes drop packets in the tunnel because FEC processing increases CPU utilization too much. We argue that the CloudLab servers used in this experiment are the bottleneck. First, the multithreading architecture allows HIRT to use more computing cores, scaling to even more offered payload. FastClick [BSM15] and RSS++ [Bar+19] can sustain more than 100 Gbps traffic while the link capacity between the CloudLab nodes should sustain 100 Gbps. We believe that with more advanced servers featuring more CPU cores for processing, one could push the limits of HIRT beyond 100 Gbps. Additionally, HIRT supports only 10 Gbps less traffic compared to the *Baseline* curve to recover 3 % of losses. With fewer losses, we expect HIRT to sustain

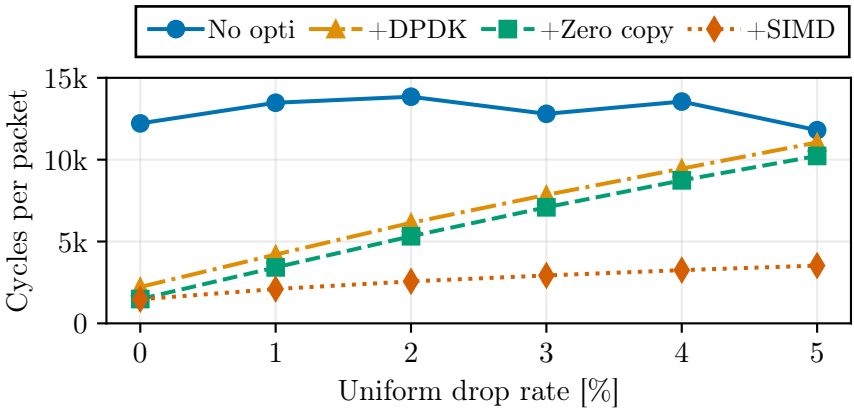


Figure 2.11: Impact of the optimizations on processing.

even more traffic and close the gap compared to the baseline. The drop rate of 3 % is chosen to understand the performance limits of HIRT. When the server keeps increasing the offered load to > 80 Gbps, we observe that HIRT provides more packets to the tunnel egress than the Baseline, with and without losses. Again, we observe that the machines used in this experiment cannot sustain such an amount of traffic. By recovering some of these losses using FEC, HIRT slightly improves throughput compared to the Baseline, even though it forwards only 55 % of the input traffic.

Impact of the optimizations. Figure 2.11 shows the average number of CPU cycles per packets of HIRT for the decoding operation with the optimizations mentioned in Section 2.6. We increase the uniform drop rate from 0 % to 5 %. Each subsequent curve adds an optimization over the previous one; e.g., the `+SIMD` curve results from enabling `DPDK`, `Zero Copy`, and `SIMD`. Without optimization, we observe that the number of CPU cycles remains constant as the uniform drop rate increases, while we would expect it to increase. This indicates that without optimization, the implementation struggles to sustain high traffic even without losses. DPDK provides the biggest performance boost, as already established by previous research [BSM15]. Then, we correctly observe that the number of cycles increases with the drop rate, as more work is required to recover lost packets. The technique that enables zero-copy delivers an almost constant 600-cycle improvement. This is independent of the number of losses, since the zero-copy applies only to received packets. Using SIMD instructions provides a strong performance gain under heavy losses. With no packet drop, there is nearly no packet recovery, and SIMD instructions do not accelerate processing. However, as the drop rate increases, the speedup approaches 3×, with 5 % of packets being dropped, because more linear systems must be solved. **With all optimizations enabled, the num-**

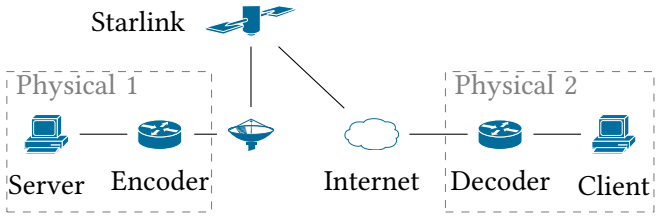


Figure 2.12: Experimental setup using the Starlink access. The *Server* and *Encoder* are run on the same physical machine using different network namespaces; similarly for *Client* and *Decoder*. The *Encoder* and *Decoder* are respectively the *CE* and *PE*.

ber of CPU cycles decreases by a factor between 3 and 9, depending on the loss percentage.

2.8 Evaluation in a real network

We demonstrate the benefits of HIRT in a real-world environment using the Starlink [STARLINK] communication system with HTTP and NFS benchmarks. Starlink is a modern Low Earth Orbit (LEO) satellite communication system that aims to provide high connectivity. Starlink gained much interest in the research community [Mic+22b; Kas+22; Ma+23; Per+22]. More particularly, measurements have shown that this medium generates losses that are *not induced by congestion* [Mic+22b]. The study measured a mean loss ratio of ~0.45 %, with bursts of losses of at most 9 packets in 75 % of measurements under light traffic. These losses *can* be recovered by an FEC algorithm, as the overhead induced by the repair packets should not contribute to increasing the number of data losses in the network. Previous works already proved the benefits of FEC in QUIC above Starlink [Mic23; MB24].

Figure 2.12 illustrates the experimental setup. The server and the encoder run on the same machine with separate network namespaces; the same holds for the decoder and the client. The encoder is directly connected to the Starlink access point; the decoder is connected to our university campus network. The RTT is ~50 ms, and we found similar loss conditions to [Mic+22b] since we share the same Starlink testbed. Since the *Server* is only reachable through Starlink, the traffic sent by the *Client* is also subject to the erasures of the medium. We only run HIRT in the *Server* → *Client* direction, as this represents the highest traffic flow.

2.8.1 HTTP requests over Starlink

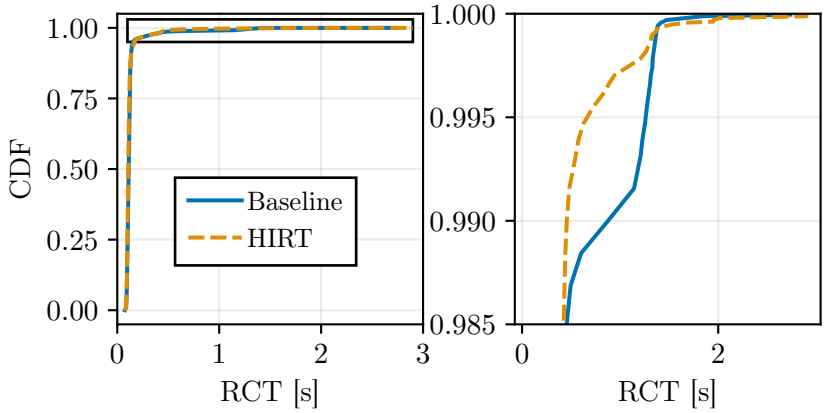
We show how HIRT transparently protects VPN traffic over Starlink. Such service can be provided to any VPN user to increase the reliability of the ser-

vice, independently of their protocols and applications. Additionally, using a VPN inside HIRT shows that the tunnel can recover losses without accessing the protected data. We use the Linux kernel implementation of WireGuard [Don17]. We start an HTTP server on the *Server*, and run a wrk [WRK] benchmark for 20 minutes. Wrk is an HTTP benchmarking tool designed to evaluate a server's responsiveness. We analyze the impact of the losses on the request completion time (RCT). Each benchmark consists of 5 parallel request flows to avoid congesting the access link.

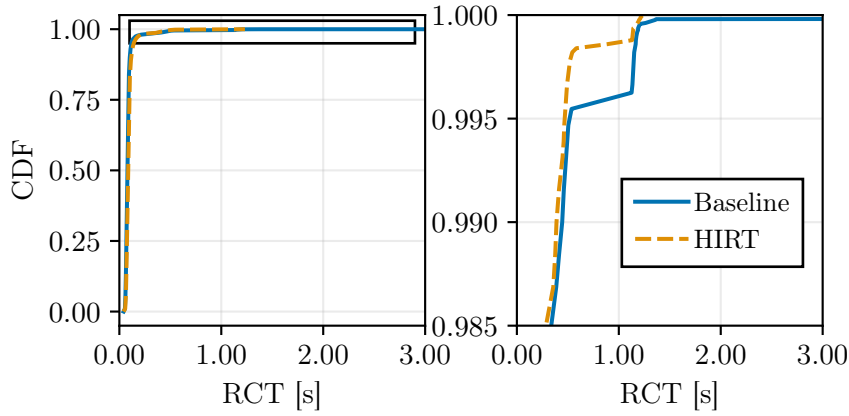
HIRT reduces the last percentile of 10 kB HTTP request completion times by a factor of 2. Figure 2.13 shows the distribution of the request completion time (RCT) for responses of 10 kB, i.e., responses that can be sent within a single round-trip. We compare the RCT between HIRT and a baseline solution only relying on retransmissions (the *Baseline* curve). Figure 2.13a uses TCP as the transport protocol. As Starlink exposes <1 % losses, the impact of packet erasure only appears around the last percentile of the distribution. For lower percentiles, we observe a similar RCT for the baseline and HIRT. The *Baseline* curve suffers from the losses, and the retransmissions increase the request completion time up to more than 1 s for almost 1 % of these requests, whereas the median over all data is ~100 ms. As expected, HIRT decreases the RCT for the 99^{th} percentile from 1 s to 500 ms. We measured an overhead of 9.25 % with HIRT. As we do not fully utilize the link, we do not expect results to vary between upload and download Starlink access.

HIRT protects traffic independently of the underlying protocols. We also used a version of wrk [Bar+21] which uses the `picoquic` [23] implementation of QUIC [RFC9000] instead of TCP as the underlying transport for the HTTP requests. Figure 2.13b shows the results. Again, HIRT does not harm the mean RCT, while improving the last percentile. HIRT decreases the 99.5^{th} RCT from 1 s to 500 ms. Both the QUIC and TCP servers used CUBIC as their congestion controller, which reacts to the observed losses by reducing the congestion window. However, for small HTTP responses such as 10 kB, these losses do not harmfully impact this window. The initial congestion window of the Linux TCP implementation is 10 packets, which is sufficient to transmit all the response within a single flight. In this case, packet losses increase the request completion time by 1 to a few RTTs, but no subsequent packets will be affected by a congestion window decrease. Next, we analyse how these packet erasures impact longer transmissions, and how HIRT contributes to improving the RCT.

By hiding the losses from the transport protocol, HIRT decreases the median RCT of longer HTTP responses by 20 % over Starlink. Longer HTTP responses involving multiple packet flights are more impacted by packet losses when the transport protocol uses a loss-based congestion controller. Improvements in the loss recovery mechanism and its impact on



(a) Wrk benchmark using TCP as the transport protocol.



(b) Wrk benchmark using QUIC as the transport protocol.

Figure 2.13: HTTP request completion time CDF for 10 kB responses. For each sub-Figure, the left part shows the global CDF; the right-hand side focuses on the last percentile (i.e., the framed part from the left-hand side).

the congestion control have been added to TCP [RFC5681]. However, loss-based congestion control algorithms use packet losses as an indicator of congestion. The erasures caused by Starlink can negatively affect the sender's congestion window, even if they are *not* due to congestion.

Figure 2.14 shows the impact of HIRT on the request completion time of TCP/HTTP responses of 1 MB (left sub-Figure). Both endpoints use CUBIC. In contrast to the 10 kB requests, we observe a positive impact of FEC across all experiments, not just the last percentile, even though Starlink losses affect fewer than 1 % of packets. With HIRT, the median is lower (1.9 s compared to 2.36 s, 18 % improvement). By recovering erased packets at the network layer, the TCP congestion controller is unaware of these losses. Thus, it avoids de-

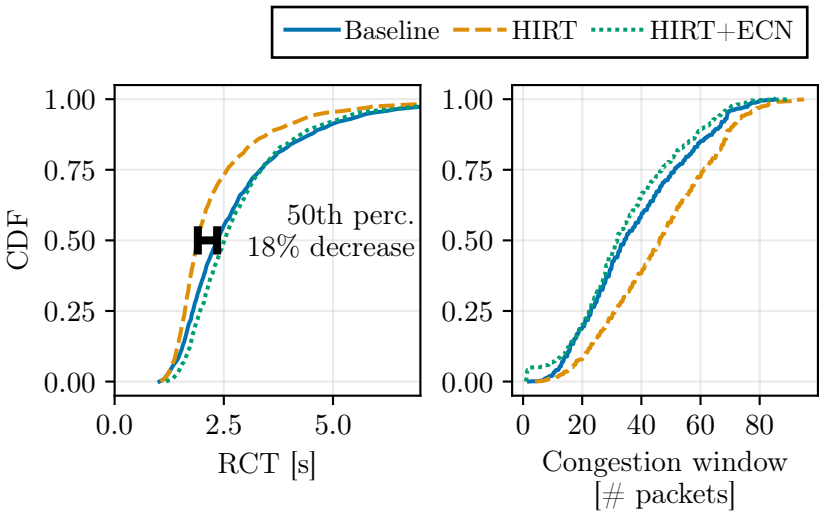


Figure 2.14: HTTP request completion time (left) and TCP congestion window (right) for 1 MB responses.

creasing its congestion window. This increases the overall throughput compared to the baseline. Figure 2.14 also reports the congestion window (CWND) of the server at the end of each HTTP response. It remains higher using FEC because the recovered packets prevent CUBIC from decreasing its window. Finally, we expect the main outcomes from Figure 2.14 to remain identical with QUIC instead of TCP, as well as with longer HTTP responses.

CUBIC vs BBR. Since CUBIC [RFC9438] is a loss-based congestion control algorithm (CCA), hiding losses has a positive impact on the evolution of the source bit rate. For shorter requests (Figure 2.13), we do not expect a difference between CUBIC and BBR [CSB25] as congestion control algorithms since the whole response fits in the first packet flight, thereby still having benefits using HIRT. For longer requests, we expect less significant results with BBR since the algorithm does not primarily use losses as a congestion indicator. Still, we believe that HIRT brings improvement by recovering erased packets, especially in the case of BBRv3, which includes packet losses in its model of the network.

By understanding Starlink's characteristics, we know that a part of these observed losses is not due to congestion, allowing the transport congestion controller to make better decisions. However, as discussed in Section 2.3, one cannot always guarantee that losses are transmission errors. To alleviate this situation, we run HIRT and mark recovered packets with the Explicit Congestion Notification (ECN) flags [RFC3168]. With this notification, the tunnel still recovers packets but no longer hides losses to end-hosts. This is represented

by the HIRT+ECN curve from Figure 2.14. Both the RCT and the CWND are similar to the baseline. We also notice that the RCT is marginally higher when ECN is activated. We believe that the 5.4 Linux TCP implementation used by the server is more optimized to handle packet losses than ECN-flagged packets. The TCP stack triggers a congestion-reduction phase upon each ECN notification received from the receiver, and when HIRT marks multiple recovered packets with CE across successive RTTs, this leads to repeated window reductions that would not occur with loss-based recovery assisted by SACK. However, we note that for the last 10 percentiles, HIRT+ECN improves the latency by ~200 ms compared to the baseline.

2.8.2 File system benchmark

Network File System (NFS) is an important application for remote workers who need to access files on enterprise servers. As these applications require interactions with humans, they are latency-sensitive. We focus on NFS benchmarks as representative of remote file access applications. Increased latency hinders the user experience. FEC protection can improve tail latency for these applications. However, due to the low traffic and the human interaction, implementing FEC in the application or relying on the transport layer would require a huge overhead to recover from lost packets. Moreover, NFS can use TCP or UDP. With HIRT, applications benefit from traffic aggregation and from the FEC protection independently of the transport protocol.

We show how increased latency due to Starlink losses impacts an NFS application. We consider this use-case realistic, as Starlink aims to provide internet connectivity to remote places in the world. We start an NFS server on the machine in our university campus and start a `fio` [FIO] benchmark on the *Client* from Figure 2.12. `Fio` executes sequential read and write operations using NFS and measures the latency of each request. We run the benchmark for 20 min with random read/write sequences of 5 MB files and a block size of 4 kB. The objective is not to objectively evaluate the NFS disk capabilities, but to see the relative impact of losses and HIRT on the latency. Figure 2.15 shows the CDF distribution of the latency of each read (left) and write (right) request from the benchmark. The improvements of HIRT are significant, **as more than 97 % of the samples present a latency below 10 seconds. Without HIRT, only 75 % of the samples are completed within this time**. For the 99^{th} percentile, HIRT improves the latency by 14 %.

2.9 Simulation results

This section provides a more thorough analysis of HIRT with the state of the art through simulations, namely highlighting the benefits of network-

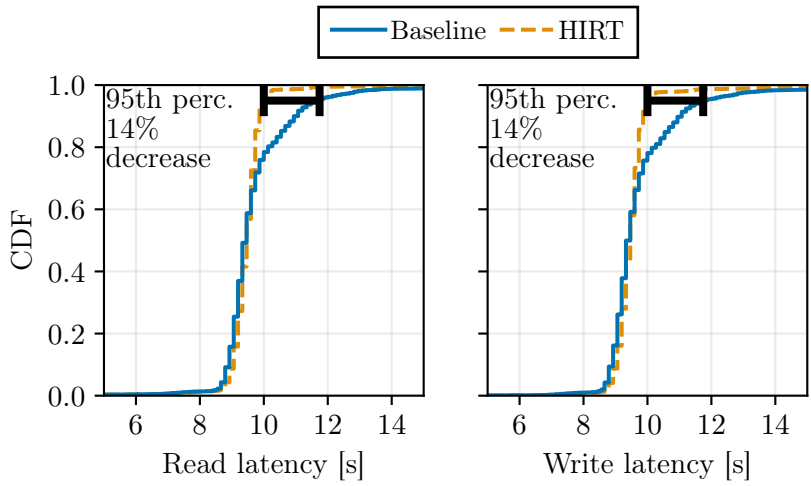


Figure 2.15: Read (left) and write (right) latency results of the `fio` benchmark.

layer FEC against transport-layer FEC (Section 2.9.1) and comparing HIRT with Maelstrom [Bal+11] (Section 2.9.2). We build an FEC packet simulator composed of five nodes, using the same topology as in Section 2.7.2, for a total of ~3000 LoC. The source code of the simulator is also publicly available. We reuse the RLC library used inside FastClick for HIRT. The *Dropper* node simulates packet erasures, both using a simple uniform drop pattern and the Gilbert-Elliott model to generate burst losses. These losses are reproducible across experiments using pseudo-random generators. To explore various loss scenarios without introducing bias in parameter selection, we again use an experimental design [Kir12] approach with 99 points when using the Gilbert-Elliott drop model, with the ranges $p \in [0.0001, 0.01]$ and $r \in [0.10, 0.50]$. For each experiment, we simulate the transmission of 50,000 packets to limit each experiment to 1-minute runs; longer transmissions do not yield different results.

2.9.1 Benefits of network-layer FEC against transport-layer FEC

We first highlight the benefits of network-layer FEC against transport-layer FEC using our simulator, i.e., we show that *aggregating packets from distinct flows* improves the recovery capabilities of an FEC system, as discussed in Section 2.3. The simulator generates packets at the ingress of the *Encoder* across 10 flows, simulating 10 concurrent transport communications within the same tunnel. The 10 flows send packets at (pseudo-)random times, with uniform random probability. We implement two variants of HIRT in the sim-

ulator:

1. *Network-layer FEC (NL-HIRT)*: HIRT, offering FEC at the network-layer, as discussed in this chapter;

2. *Transport-layer FEC (TL-FEC)*: a tweaked variant of HIRT, protecting each flow individually. In a nutshell, TL-FEC is similar to implementing HIRT at the transport layer.

We report the results in Figure 2.16.

NL-HIRT recovers erased packets much faster than TL-FEC. The left part of Figure 2.16 shows the ratio of recovered packets across the 99 experiments. The transport-layer FEC system recovers more erasures in 25 % of the experiments, but network-layer FEC outperforms the first approach in the remaining runs. We expected such results since both approaches use the same RLC coding scheme underneath; in the case of TL-FEC, each flow carries sufficient source symbols ($\sim$ 5000 each) to recover the losses inside each independent tunnel.

The right-hand side of Figure 2.16 exhibits the real benefits of NL-HIRT. For each erased and recovered packet, we count the introduced *jitter* relative to other packets from the same flow, since this packet is recovered only after enough source and repair symbols are received by the *Decoder*. Because a single tunnel aggregates the ten flows when generating repair symbols, NL-HIRT recovers losses faster than per-flow FEC tunnels, which can only recover the lost packet once enough symbols from *their own* flow arrive.

2.9.2 Comparing with Maelstrom

Maelstrom [Bal+11] is the closest related work to HIRT. It suggests deploying proxies between DCs and WANs and injecting FEC packets between unmodified packets at the network layer. It proposes encoding packets across multiple layers. For example, the first layer encodes every packet, while subsequent layers, e.g., with an interleaving of 4, encode 4 streams of 1 packet every 4 packets. The idea is to allow recovery of dropped bursts. All layers use the XOR coding scheme to generate the repair symbols. We note the following differences between Maelstrom and HIRT: (i) Maelstrom suggests constant-rate redundancy generation inside the tunnel. It does not adapt to the network condition. As a result, Maelstrom may send too many or too few repair packets, respectively causing bandwidth waste or impossibility to recover the majority of lost source symbols; (ii) Maelstrom design does not focus on high-speed packet processing and could not sustain more than 700 Mbps [Bal+11] of protected traffic in 2011. The throughput should be higher on more recent machines, but we could not test it because the source code is not available.

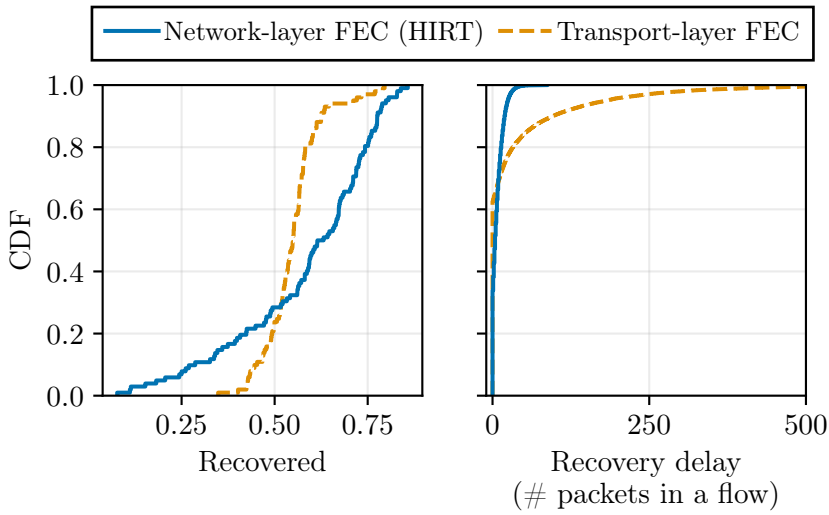


Figure 2.16: By aggregating flows together when generating repair symbols, network-layer FEC (modelling HIRT) recovers lost symbols more quickly than transport-layer FEC (i.e., per-flow HIRT).

In this section, we compare the recovery capabilities of HIRT with those of Maelstrom. We added the Maelstrom encoder and decoder components to our simulator. We use a single-source flow because we focus on comparing two distinct network-layer FEC mechanisms. We tune Maelstrom with different interleaves and window sizes to trade off overhead for recovery capabilities, using the best interleaves from the original paper [Bal+11]. In Figure 2.17 and 2.18, *M:X-Y-Z:W* means Maelstrom with interleaves $I = (X, Y, Z)$ and a window W, similarly to the notation in Maelstrom's paper [Bal+11]. For example, *M:1-3-6:4* means that Maelstrom uses three layers of packet encoding, each with a window of 4 symbols:

- *1*: each (non-overlapping) window of 4 consecutive symbols generates a new repair symbol;
- *3*: the second layer groups symbols with an identifier %3 == 0 by windows of 4 symbols together, %3 == 1 and %3 == 2 separately;
- *6*: similarly, symbols with identifiers %6 == 0 form another layer of source symbols, %6 == 1, ...

For an interleave Y, there are Y parallel windows of size W containing symbols with gaps of Y symbols.

Uniform drop model. We first compare HIRT and Maelstrom by increasing the uniform drop rate in Figure 2.17. We measure the overhead,

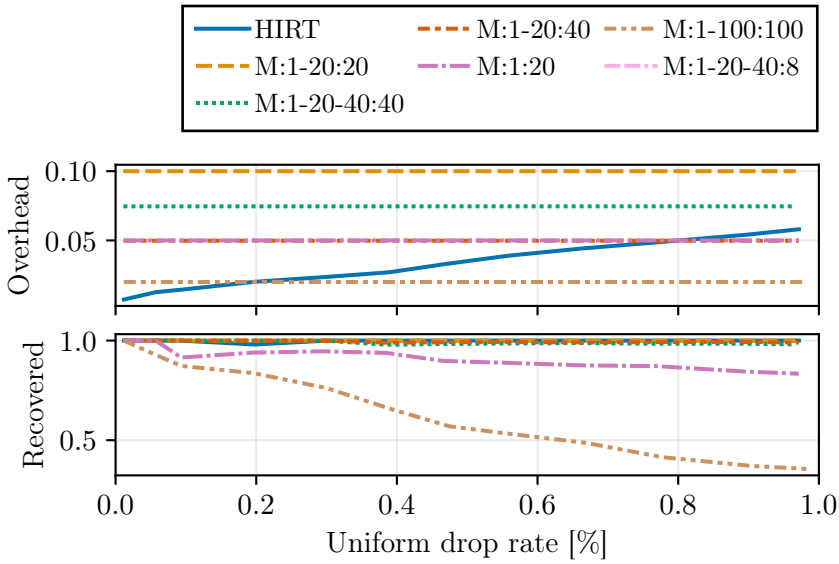


Figure 2.17: Uniform drop model simulation.

i.e., the ratio of repair packets injected to the network compared to the number of source symbols (upper sub-Figure), and the ratio of recovered packets, i.e., the number of recovered symbols with respect to the number of losses (lower sub-Figure). Maelstrom does not adapt to the uniform drop rate since it uses static interleaving. Figure 2.17 shows that HIRT correctly adapts to the observed losses, and provides lower overhead than Maelstrom while still recovering erasures. For example, with a drop rate of 0.5 %, HIRT recovers all erased symbols with an overhead of 3.3 % while *M:1-20:20* only recovers 89 % of them for an overhead of 5 %. With more layers (e.g., *M:1-20-40:20*), Maelstrom recovers almost all lost symbols at the cost of a higher and constant overhead compared to HIRT. Even if RLC is more computationally intensive, its recovery capabilities outperform Maelstrom's method.

Gilbert-Elliott model. Figure 2.18 presents the CDFs of the overhead (left) and ratio of recovered packets (right) using the experimental design approach with the Gilbert-Elliott drop model. The impact of burst losses on Maelstrom's performance is significant: with $I = (1, 20)$ and a window of 20, more than 50 % of the experiments recover less than 75 % of the erased packets. On the other hand, the adaptive algorithm allows HIRT to correctly adapt to the loss rate. More than 95 % of erased packets are recovered in 80 % of the experiments. Figure 2.18 also presents how Maelstrom performs with $I = (1, 20, 40)$ and a window of 8 packets, values similar to the original paper [Bal+11]. In that setup, the overhead is 34 %, which is constantly above

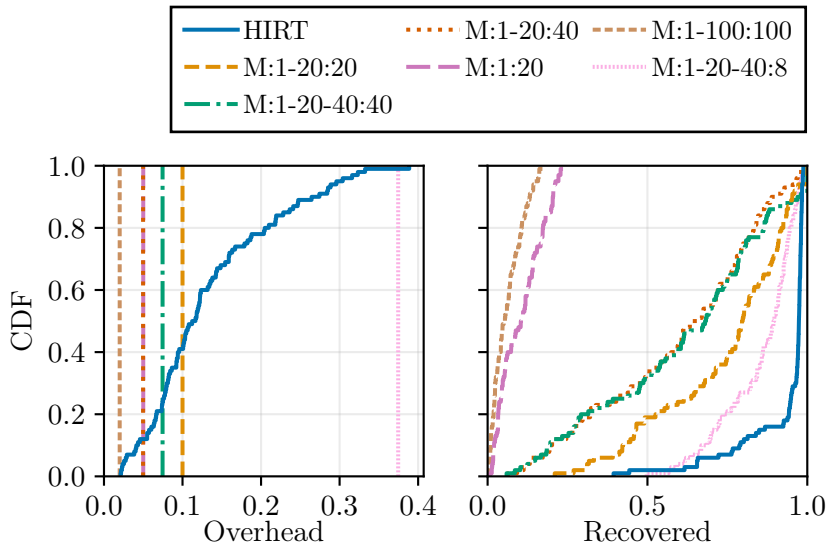


Figure 2.18: Simulation with the Gilbert-Elliott drop model using experimental design with 99 points. *M:X-Y-Z:W* means Maelstrom with interleaves $I = (X, Y, Z)$ and a window W.

that of HIRT. However, the ratio of recovered packets of Maelstrom remains almost always lower than HIRT.

Figure 2.19 shows results where we approximate the loss model of Starlink [Mic+22b]; we set $r = \frac{1}{3}$ and $p = 0.00151$. Again, HIRT outperforms Maelstrom. **With its adaptive scheduling and the Convolutional RLC erasure code, HIRT recovers many more erased packets than Maelstrom while ensuring lower overhead**. As the implementation of Maelstrom is not publicly available, we could not compare its performance at high speed. Even if we could expect a higher throughput than 700 Mbps [Bal+11] nowadays, we still believe that the architecture of HIRT and the optimizations done would sustain a higher throughput than Maelstrom.

2.10 Related work

This section reviews related state-of-the-art work to improve network-layer communication latency.

Network-layer coding. Sundararajan *et al.* [Sun+11] propose a network-coding extension of TCP. The FEC mechanism is implemented between TCP and the network layer, and intermediate routers can perform packet coding to recover from packet erasures. Our approach is completely agnostic of the transport layer of the protected packets. It leverages SRv6 network program-

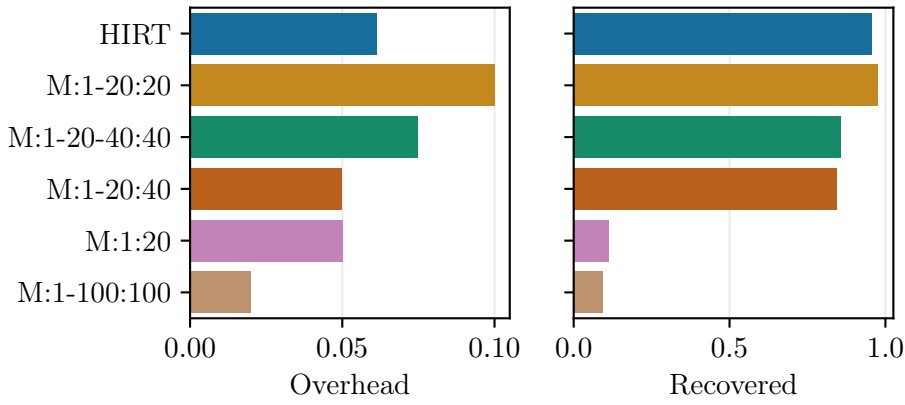


Figure 2.19: Simulation with the Gilbert-Elliott drop model, fitting Starlink's loss rate. *M:X-Y-Z:W* means Maelstrom with interleaves $I = (X, Y, Z)$ and a window *W*

ming [RFC8986] rather than extending the TCP stack to deploy the solution. Moreover, our work focuses on a design and implementation that can protect traffic at high speeds.

Loss recovery. Another option to mitigate losses without FEC is to send *a priori* several copies of the same packet. Duplicating the traffic would increase too much the bandwidth consumption and CPU usage. Selective Loss Prevention [ZY21] proposes a new algorithm that duplicates only selected packets in a TCP flow, such as those with the SYN and PSH set. It reduces overhead and improves performance in a lossy environment. HIRT suggests an underlying-layer-independent tunnel that can recover any packet in the tunnel. The FEC redundancy provides protection across a range of packets at the cost of higher CPU usage for encoding and decoding the repair symbols.

AC-RLNC [Coh+20b] proposes an adaptive, causal network coding algorithm with feedback for multipath and multi-hop communications. Using FEC, AC-RLNC adapts its coding rate by leveraging feedback from the end-host every RTT. Repair symbols are sent *a priori* using this estimation, and *a posteriori* with feedback to reach reliability. No implementation of AC-RLNC could be found for performance comparison. Because HIRT includes a feedback mechanism, extending our solution to support AC-RLNC is theoretically possible, though it would require meticulous tweaking to provide high-speed packet processing.

Congestion avoidance. Microbursts are known causes of bursty data center losses [BAM10]. Deflection is a mechanism that temporarily changes a packet's port to reroute it if it enters a full buffer, preventing packet loss from such microbursts. Deflection adds reordering as some packets may be

re-routed and not subsequent packets of the same flow. Vertigo [ASG21] proposes a transport-independent extension in end-host stacks to recover the original packet order before handing over the packet flow to the transport protocol. Even if this is not directly related to HIRT, it shows that both packet reordering and packet loss occur in data centers. We suggest an approach that can be combined with Vertigo.

Active networking. Active Congestion Control (ACC) leverages active networking to indicate to each router in the network how to respond to congestion [Fab98]. By actively including routers in congestion control, ACC reduces reaction time in wide-area networks. Leveraging a similar mechanism could improve HIRT's resilience to congestion. However, it would require per-flow state to work well with aggregated traffic.

An architecture for active networking suggested the use of predefined functions that could be called directly by the application [BCZ97]. The authors then proposed implementing congestion-related functions in the network and allowing applications to determine how routers should react when congestion arises. Applications cannot create new functions within the network; they can only use the provided API. This approach is similar to SRv6 network programming [RFC8986]. HIRT leverages the network programming paradigm to enable applications to dynamically request protection within the tunnel.

Data Center Networks (DCN). In ReWAN [HD15], the authors propose a similar approach using an inter-DC scheme that provides recovery-as-a-service. End-hosts notify losses and get recovered packets. Most practical details, such as which source packets the repair service should snoop on, are left for future work. In a later proposal [HRD17], the authors measured inter-DC loss characteristics and argued that a combination of one-hop detour routing and FEC might allow recovery from inter-DC losses. This scenario is enabled by our proposal.

CloudBurst [Zen+21] proposes to combine the use of FEC and multipath in datacenter networks. They implement a transport library that allows users to transmit messages over multiple paths within a datacenter network. Our proposal achieves a similar goal without host and application modification.

Satellite communication. OpenSAND [Dub+17] is a satellite emulator protecting data with datalink-layer FEC. HIRT differs by protecting entire satellite constellation paths with a deployable high-speed implementation, rather than an emulator. Recent work suggests adapting existing congestion control algorithms for satellite communication [Kam+24; Lai+25]. HIRT can still be combined with such modified congestion control algorithms to recover from packet losses instead of relying on retransmissions.

eBPF implementation. We also considered an eBPF implementation of network-layer FEC in previous work [NMB21]. Due to limitations in the

eBPF verifier at the time, it was impossible to implement Random Linear Code (RLC) in eBPF byte code. The generation of each repair symbol and the creation and solution of linear systems of equations during recovery required regular communication with a user-space process, which slowed the approach's performance. This prototype could only protect traffic at a few megabits per second.

2.11 Discussion

In this chapter, we designed HIRT, a high-speed, robust tunnel that uses Forward Erasure Correction (FEC) at the network layer. IPv6 Segment Routing and its network programming [RFC8986] allow operators to dynamically deploy HIRT to improve the latency of their applications by recovering packet losses without retransmissions. More precisely, we carefully designed and implemented HIRT to protect traffic at high speeds and to optimize redundancy based on feedback from the tunnel egress. Our benchmarks highlighted the system's potential and limits. We experimentally showed the benefits of the tunnel above the Starlink access network, which exposes losses not induced by congestion [Mic+22b].

While transport-layer FEC brings knowledge from the application, we believe that network-layer FEC systems, such as HIRT, provide transport-agnostic protection to improve applications' latency, especially with the advent of LEO satellite constellations. Shortly after the publication of the paper derived from this chapter, Cisco also released an article about their use of network-layer FEC to improve latency over Starlink [cisco24]. Their solution uses simple XOR algorithms. While we cannot formally compare HIRT with Cisco's solution, this article shows the interest among industrial companies in systems similar to ours.

Improvements to this work. We evaluated HIRT on a real network (Starlink) in multiple scenarios. However, we argue that more evaluation could strengthen the benefits highlighted in this chapter. First, Section 2.9.1 exhibits the benefits of network-layer against transport-layer FEC through simulations. Real measurements also account for the *recovery delay*, i.e., the time required to recover a lost packet, which is a more interesting metric regarding user experience. Instead of relying on simulations, this work could also consider evaluating HIRT against real implementations of transport-layer FEC [Mic+22a; MB24]. We do believe that the benefits of HIRT from our real-network evaluation (Section 2.8) primarily come from traffic aggregation between multiple flows – especially to improve the tail latency request completion time of short HTTP requests.

Second, the evaluation focused on CUBIC, a loss-based congestion control algorithm (CCA). The losses exposed by Starlink [Mic+22b] have a greater

impact when using CUBIC rather than model-based algorithms, which do not primarily rely on losses. F. Michel observed benefits from using FEC in QUIC and the BBRv2 congestion control algorithm over Starlink [Mic23]. Still, improvements to this chapter should include evaluations using BBR next to CUBIC. We do not expect the benefits of HIRT to change for short HTTP requests and NFS benchmarks, since these involve short query-response sequences that benefit from traffic aggregation at the network layer. For longer HTTP responses, we expect BBR to outperform CUBIC in terms of request completion time since Starlink's losses impact less the transmission rate of the source.

Third, L4S-enabled networks [RFC9330] present interesting synergies with HIRT. L4S networks signal congestion events *before* dropping packets, thereby enabling end-hosts to adjust their sending rates before generating losses in the network. Deploying HIRT in L4S-enabled networks could further reduce application latency.

Conclusion. This chapter concludes the first part of this thesis. As discussed in the introduction, network over-provisioning leads to spare bandwidth that we use with HIRT to improve the latency of applications. In the next chapter, we tackle the main reason we encounter this network over-provisioning: events of large one-to-many distribution of information, where numerous clients request the same content at (nearly) the same time. We will reconsider *multicast* distribution, a natural approach to these distribution patterns. More precisely, we will extend the QUIC [RFC9000] protocol to provide reliable, secure, and flexible transport-layer multicast.

Flexicast QUIC: Taking the Best of Multicast and Unicast

3

In the previous chapter, we leveraged spare bandwidth provided by network providers' over-provisioning. Such over-provisioning is required to handle sporadic traffic peaks during popular events, such as live streaming [Ber25], which are inherently one-to-many. Multicast theoretically offers scalability in these scenarios, but deploying it across inter-domain boundaries remains difficult [Dio+00], leading content sources to use unicast protocols like TCP and QUIC instead to distribute their content on the Internet.

Multiple causes justify the lack of inter-domain multicast deployment on the Internet. We focus on the transport layer with the following statement.

> Multicast is not available everywhere, and multicast transport protocols lack unicast mechanisms, discouraging applications from deploying multicast since they must implement their own fallback methods.

In this chapter, we revisit multicast transport. Starting with the QUIC protocol [RFC9000] and its soon-to-be standardized multiple paths management extension [Liu+26], we design, implement, and evaluate *Flexicast QUIC*, which provides *flexible multicast* transport with seamless coordination with unicast for the source, the receivers, and application maintainers. With Flexicast QUIC, the source benefits from the scalability of multicast when available, and uses unicast for receivers without multicast support, all in the same protocol. The real gains of flexicast come from the interaction between multicast and unicast: receivers can *seamlessly switch* between them during the lifetime of the connection, depending on the quality of the communication. This chapter focuses on video streaming because it requires low latency, and live streaming constitutes the primary use-case motivating multicast communication.

Contributions. We make the following contributions:

- The design of *flexicast* and its integration in QUIC.

- The implementation of Flexicast QUIC as an extension of Cloudflare *quiche* [QUICHE].
- Benchmarks assessing that a single source can sustain >80 Gbps of traffic aggregated across 1000 receivers.
- Evaluations through a video stream use-case showing the robustness of Flexicast QUIC when the multicast network fails or underperforms for a subset of the receivers, considering the problem of heterogeneous receivers while ensuring good quality of multicast distribution.

The remainder of this chapter is organized as follows. Section 3.1 details our rethink of the features a multicast transport protocol should provide, and how we built upon the multipath capabilities of transport protocols to design flexicast. We detail the extensions to integrate flexicast into QUIC in Section 3.2 and present our first implementation in Section 3.3, pointing towards a scalable server. Section 3.4 evaluates Flexicast QUIC using both benchmarks and emulations. We conclude this chapter with a presentation of the recent works about multicast at the transport layer in Section 3.5 and a retrospective discussion in Section 3.6. The majority of the results from this chapter led to the publication of a journal paper in ACM SIGCOMM Computer Communication Review, Volume 55, Issue 2 (2025). We also more formally describe the design of Flexicast QUIC in an IETF draft [NB25a].

3.1 Flexible multicast

In the 90's, researchers developed several one-to-many-capable transport protocols [Obr02]. The Real-time Transport Protocol (RTP) [RFC3550] was initially designed to support multicast for media distribution. RMTP [Pau+97] focused on reliable delivery, but remained a research protocol that was not deployed. More recently, the NACK-Oriented Reliable Multicast (NORM) protocol [RFC5740] built on the lessons learned from these older protocols to design a scalable multicast transport protocol using negative acknowledgment for reliability. However, except for specific use-cases, NORM never attracted the attention of content distributors.

Due to the low inter-domain multicast deployment, applications cannot always guarantee that all receivers have access to multicast routing. As such, we believe that one significant limitation of existing multicast transport protocols is that they *assume* multicast connectivity. Indeed, none of the aforementioned protocols provides a seamless mechanism for applications to fall back to unicast, when required. This limitation led to the *all-or-nothing* problem of multicast: for a communication to use it, all participants *must* support it. Another issue arising from this observation is that the protocol must define

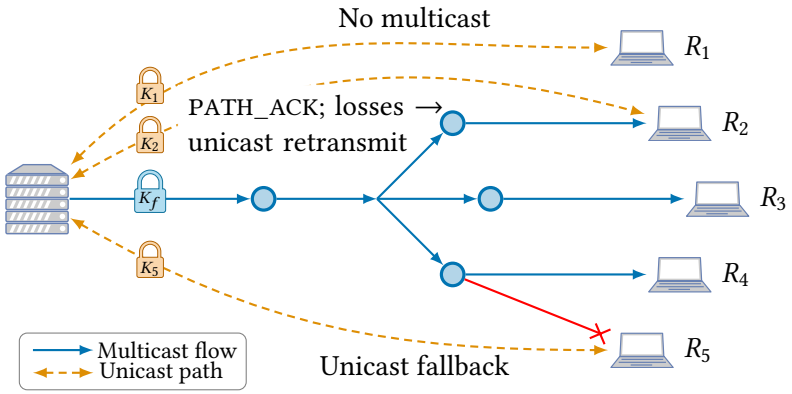


Figure 3.1: High-level overview of Flexicast QUIC. A connection combines per-receiver bidirectional unicast paths and shared unidirectional *multicast flow(s)*. The unicast paths are protected using distinct keys ($K_{\{1\to5\}}$) while the multicast flow uses a common key, K_f. Flexicast QUIC leverages Multipath QUIC [Liu+26] to manage the multicast flow as an additional path. We omit the unicast paths of the two middle receivers for clarity.

ad hoc methods to prevent a single bottleneck receiver from slowing down the entire group, such as clustering receivers into subgroups based on their reception rates [GHK02] or using multiple multicast channels [MJV96].

Starting from QUIC and Multipath. The emergence of QUIC [RFC9000] and Multipath protocols [RFC8684; Liu+26] lets us reconsider the interaction between multicast and unicast at the transport layer. QUIC provides modern reliability features [RFC9002], embeds TLS into its design [RFC9001], and, unlike TCP, decouples connection identifiers from the classical four-tuple. Additionally, QUIC runs on top of UDP, making it a suitable candidate for IP Multicast support. With Multipath, end-hosts can create multiple bidirectional paths within the same transport connection. We extend this principle by allowing one of these paths to be shared between multiple receivers simultaneously. The IP destination address of packets sent on this path can be a *multicast* address, thereby leveraging the scalability of an underlying multicast network.

This chapter introduces the concept of *flexible multicast* (*flexicast*), which is illustrated in Figure 3.1. 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 uses TLS to negotiate 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, and communicates it to all receivers via these secure unicast paths. The source uses the key K_f to encrypt packets targeting all receivers simultaneously,

thus creating a shared unidirectional flow of data that we call a *multicast flow*. Packets sent on this flow can be delivered to the receivers, either through an underlying multicast tree or using replication with IP unicast.

Multicast flow. Let us denote by m the number of receivers attached to the multicast distribution tree. If $m = N$, i.e., all receivers have multicast connectivity, 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 multicast flow. However, there are situations in which only a fraction of the receivers have joined the multicast tree. Consider that receiver R_1 cannot join the multicast tree (e.g., R_1 is in a network that disables multicast). In addition to the packets sent on the multicast flow, the source will generate and encrypt a new packet for R_1 using K_1 with the same application data.

If $0 < m \leq N$, i.e., $N - m$ receivers did not manage to join the multicast tree, the source needs to encrypt and authenticate $N - m + 1$ packets for each data chunk it sends. When $m = 0$, this method would be equivalent to delivering data over unicast, which is the current solution QUIC servers use nowadays [Hol20]. However, measurement studies [Jae+23; Tyu+22] show that generating and encrypting QUIC packets consume substantial CPU resources on servers.

Flexicast offers a more efficient and elegant solution. Considering the worst case, i.e., $m = 0$, the source could still generate a single packet and encrypt it using K_f. Instead of relying on an underlying multicast distribution tree, the source could replicate the bytes generated by the transport protocol and send them over IP unicast to each receiver. This solution is more efficient (CPU-wise) because replicating bytes and changing the IP destination address is faster than generating and encrypting new QUIC packets for each receiver. The Linux `sendmmsg` system call can efficiently perform such an action.

Flexible multicast. The source maintains per-receiver unicast paths during the lifetime of the connection. These paths offer a secure fallback mechanism whenever (i) the underlying multicast distribution tree fails (e.g., R_5 in Figure 3.1), or (ii) a receiver becomes a bottleneck, e.g., it cannot keep up with the multicast flow. Such a receiver can leave the multicast flow and seamlessly fall back to unicast within the same connection, thus offering *flexible* multicast to endpoints: multicast for its efficiency and unicast for its robustness. Similarly, the source uses these unicast paths to retransmit packets sent and lost on the multicast flow (e.g., R_2).

3.2 The design of Flexicast QUIC

This section presents the design of Flexicast QUIC (FCQUIC), which builds upon QUIC [RFC9000] and its multiple paths management extension [Liu+26];

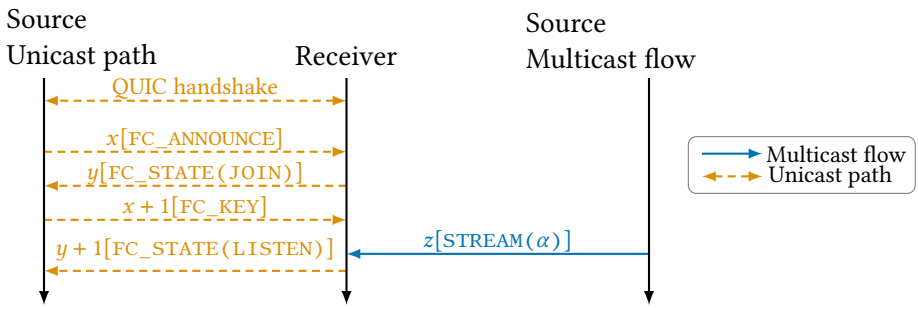


Figure 3.2: After the QUIC handshake, the source advertises its available multicast flow. If the receiver joins, the source sends the flow's decryption key. Once the receiver starts receiving packets from the multicast flow, it notifies the source through the FC_STATE(LISTEN) frame. Packets are represented as $\{pn\}[\text{FRAMES}]$ with pn being the packet number.

for simplicity, we call this extension *Multipath QUIC* (MPQUIC). FCQUIC relies on Multipath QUIC for the creation, management, and scheduling of the multicast flows. A Flexicast QUIC connection starts as a regular (Multipath) QUIC connection, besides the use of the `enable_flexicast` transport parameter. During the handshake, the source and receiver establish a first bidirectional path protected by a single set of TLS keys – the unicast path. 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 multicast 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 multicast flow.

3.2.1 Advertising a multicast flow

Figure 3.2 illustrates the first steps of a Flexicast QUIC connection, which starts with a standard QUIC handshake [RFC9000], 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 of the multicast flow. After the QUIC handshake, the source announces the available multicast flow using the FC_ANNOUNCE frame. A multicast flow is identified by a destination Connection ID (CID) chosen by the source. For clarity, we refer to the destination CID of the multicast flow as the *Flow ID* (FID). The FC_ANNOUNCE frame contains the FID, the source

and multicast IP addresses forming the (*S*, *G*) tuple, and the UDP destination port number of packets sent on the multicast flow. We assume that the receiver does not already use either the FID as a source CID or the UDP port. As each multicast flow is viewed as *a new, independent path*, it is possible to advertise multiple flows simultaneously. In the case of a video stream, each flow can convey the stream encoded in a different quality, or use layered coding [MJV96].

Joining a multicast flow. A receiver sends an FC_STATE frame with the JOIN action to the source to request joining the advertised multicast flow. At this point, the receiver creates a state for the new multicast flow. The source answers by sending the FC_KEY frame containing the shared key used to encrypt packets sent on the multicast flow. The receiver then joins the underlying multicast tree, e.g., using IGMP [RFC3376] or MLD [RFC3810]. Once the receiver starts receiving packets on the multicast flow, meaning that the underlying multicast tree is working, the receiver sends an FC_STATE frame with the LISTEN action. If multicast is unavailable, the receiver falls back to unicast. The FC* frames are exchanged over the unicast path and are thus secured using the (unicast) TLS keys negotiated during connection establishment.

3.2.2 Changes to Multipath QUIC

The current version of MPQUIC [Liu+26] uses a single cryptographic context for the entire connection, i.e., all paths use the same key. Packets are encrypted and authenticated using AEAD with a Nonce whose value depends on the path identifier, but the encryption key remains the same. FCQUIC extends MPQUIC by providing a cryptographic context for each path. The multicast 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 multicast flow to be *unidirectional* once established. Whilst this is compliant with MPQUIC's design [Liu+26], FCQUIC has specific considerations: a receiver sends *all* its packets on the unicast path and uses the multicast flow only to receive them. The source never receives packets on the multicast flow. It only sends frames intended for all receivers on the multicast flow.

Thanks to Multipath QUIC, receivers view the multicast flow as a second MPQUIC path with a different decryption key. Because MPQUIC [Liu+26] 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 multicast flow carries the *data* (e.g., STREAM) frames. The source can also retransmit frames over the unicast path to recover from losses. Receivers with a failing multicast flow can fall back to unicast to receive sub-

sequent data.

The third change involves creating a multicast 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 [Liu+26]. In MPQUIC, each new path is bound to a set of CIDs. In FCQUIC, since the destination address of the multicast 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 trusts the advertised multicast address. Additionally, because the multicast flow is unidirectional, an external attacker cannot spoof the receiver's address to send malicious data to the source.

3.2.3 Reliability mechanisms

By default, a QUIC receiver should send an ACK frame after receiving at least two ack-eliciting packets [RFC9000]. The default maximum delay between two acknowledgments is set to 25 ms [RFC9000]. Multipath QUIC extends this procedure by replacing ACK frames with PATH_ACK frames [Liu+26]. These frames include the *path 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 one initially used. MPQUIC endpoints decide the scheduling of the frames they send and retransmit [Liu+26]. For example, they can retransmit a frame previously sent and lost on path Y along path X. FCQUIC leverages Multipath to ensure full reliability by retransmitting frames either on the multicast flow or the unicast path, depending on how many receivers have reported a loss.

Acknowledgments on the unicast path. To acknowledge data received on the multicast flow, FCQUIC receivers send PATH_ACK frames on their unicast path with the *path identifier* of the multicast flow, as illustrated in Figure 3.3. To prevent ACK implosion, an FCQUIC source can advertise a *minimum ack delay* inside the FC_ANNOUNCE frame. We analyze the impact of this parameter on the scalability of FCQUIC in Section 3.4.1.

The source releases memory for a packet sent on the multicast flow once all receivers have acknowledged it, or it is considered lost [RFC9002]. If the packet is lost, the source retransmits the reliable frames on the unicast paths to receivers that did not acknowledge the corresponding packet. As such, the computed RTT on the multicast flow is a function of the *slowest* receiver to acknowledge a given packet number.

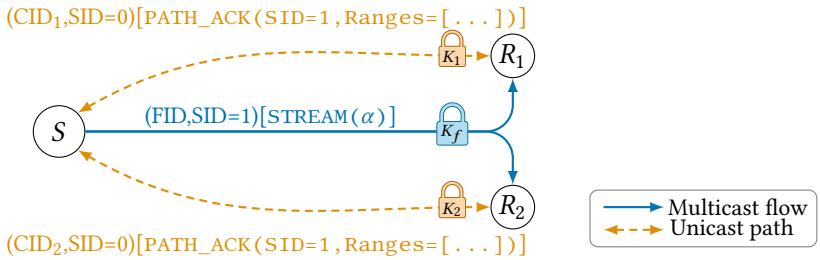


Figure 3.3: While a QUIC path is identified by a Connection ID (CID), multicast flows are identified by a Flow ID (FID). Each path/flow uses its own packet number space ID (SID). Receivers (R_1 and R_2) send acknowledgments using the PATH_ACK frame on their unicast path, using the *space identifier* of the path the acknowledgments refer to.

Congestion control. Because the multicast flow is shared, its throughput depends on all receivers. The FCQUIC source maintains a per-receiver congestion state for the data sent on the multicast flow. These congestion states are adapted using the per-receiver acknowledgments, leveraging existing unicast congestion control algorithms such as CUBIC [RFC9438] to determine the appropriate sending rate. This initial version of FCQUIC sets the throughput of the multicast flow to the minimum rate among all active receivers for that flow. As such, the source adapts its transmission rate to the slowest receiver. The Multipath QUIC design uses distinct congestion control states for each path. Thanks to this, the congestion state of the multicast flow is independent of the unicast path state since the source only uses the PATH_ACK frames with the *space identifier* of the multicast flow to determine its sending rate. The source can remove receivers with insufficient performance (e.g., those with a too-low reception bit rate) from the multicast flow and continue sending data through their unicast path. Managing the multicast flow ensures a minimum level of quality of experience and enables the source to handle potentially malicious receivers.

Flow control. The FCQUIC source manages flow control on a per-receiver basis. It sets the limits to the minimum among all active receivers, potentially removing from the multicast flow members whose flow control boundaries do not increase sufficiently fast to prevent blockage for the entire group. Reusing Multipath QUIC semantics, receivers send flow control updates via MAX_DATA and MAX_STREAM_DATA on their unicast path to the source. The flow control mechanism of QUIC requires the updated limits to be monotonically increasing, i.e., new limits represent the cumulative number of bytes the receiver is willing to accept. When joining the multicast flow, new FCQUIC receivers are notified of the current flow control limits in the FC_ANNOUNCE frame, allowing them to adjust their state to start at that offset and use incremental updates thereafter.

3.2.4 Managing the multicast flow

Both the Flexicast QUIC source and receivers manage the status of the multicast flow using the FC_STATE frame. Receivers use the LEAVE action to leave the multicast flow. Upon reception, the source falls back on unicast for this receiver, using the unicast path and standard QUIC for data delivery. Leaving the multicast flow is identical to removing an existing path in Multipath QUIC [Liu+26]. The source can also *unilaterally* eject a receiver from the multicast flow by sending an FC_STATE frame with the LEAVE action. Once the source sends this frame, it no longer considers the receiver in the multicast flow and falls back to unicast. For example, this event can occur if the receiver reports too many losses relative to the other receivers, meaning the multicast flow would underperform if this receiver were kept.

There are also situations where a working multicast 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 3.4.2 explores this scenario and shows how a Flexicast QUIC source deals with such failures.

3.3 Scalable Implementation of Flexicast QUIC

We implement FCQUIC inside the multipath branch [QUICHE-MULTIPATH] of Cloudflare *quiche* [QUICHE] written in the Rust programming language. We chose *quiche* due to our familiarity with the implementation compared to other stacks supporting the multipath extension; *quiche* is production-ready, and provides consistent goodput compared to the other stacks of the QUIC landscape [Kem+24]. Our implementation adds ~10,000 lines of code (LoC) and 5000 LoC of tests. Our multithreaded applications based on `tokio` [TOKIO] required ~4500 LoC. The FCQUIC receiver application is relatively similar to a Multipath QUIC equivalent since the multicast flow is considered as a *new* unidirectional path. When it receives a packet with the multicast 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 multicast flow.

The main challenge is to design a scalable architecture for the Flexicast QUIC source with a shared multicast flow, where information from each receiver affects the multicast flow's state while ensuring scalability for large groups. This section describes how we implemented the communication between the per-receiver unicast paths and the shared multicast flow on the source to ensure scalability. Figure 3.4 presents an overview of this architecture, considering four receivers listening to the same content. Our architecture comprises 4 elements.

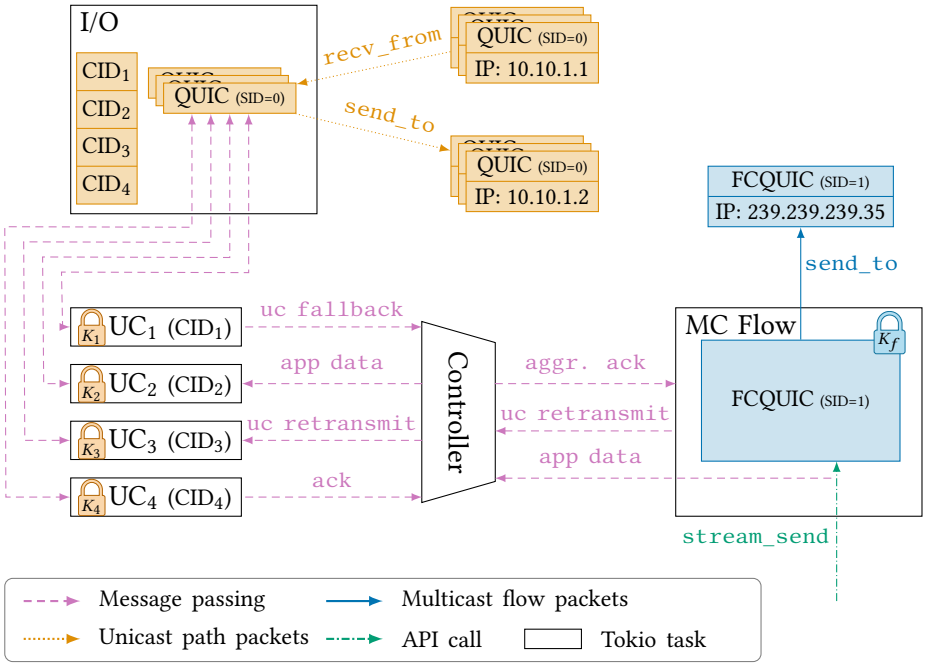


Figure 3.4: Overview of our implementation of the Flexicast QUIC source.

I/O. The *I/O* module receives and sends unicast packets. When receiving a new packet, the module verifies the destination CID. If the CID maps to an existing connection, the packet is forwarded to the associated unicast path; otherwise, it 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 3.4, they correspond to the packet number space ID (SID) 0, the value of the first path created upon connection establishment [Liu+26]. 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 multicast flow. The *UC* module checks the liveness of the multicast flow and, when relevant, notifies the controller that the corresponding receiver should fall back to 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.

Multicast flow. The multicast flow (*MC Flow*) receives data from the application, either using streams or datagrams, and generates packets on the multicast flow, using the shared key (K_f) and the Flexicast flow ID (FID). It

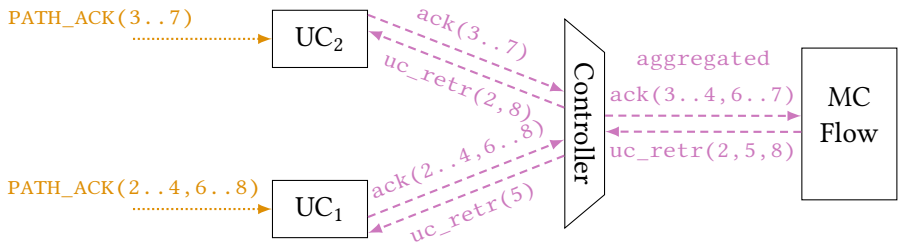


Figure 3.5: The controller aggregates acknowledgments received on the unicast path before sending them to the multicast flow module. It delegates unicast retransmissions to the controller, which dispatches them to the unicast path based on the per-receiver acknowledgments.

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 multicast flow. Its role is to aggregate information (e.g., acknowledgments) from the receivers and dispatch unicast retransmissions to them. This module provides scalability by handling per-receiver information and communicating only aggregated updates to the *MC flow* module. 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 uses the `tokio` [TOKIO] 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 [FHS24] and designing specific protocols to communicate using the same control messages as designed in this chapter.

Scalable reliability. All active receivers must acknowledge each packet sent on the multicast flow before the source can release it from memory. This raises a challenge regarding how to manage acknowledgments while ensuring scalability. The *Controller* aggregates acknowledgments from each receiver and sends a single acknowledgment for each packet to the multicast flow module. It then distributes the unicast retransmissions using these acknowledgments. Figure 3.5 shows an example of acknowledgment aggregation and unicast retransmission with two receivers. In this example, the first UC_1 instance receives acknowledgments through the PATH_ACK frame from the receiver for packets 2 to 4 and 6 to 8; UC_2 receives acknowledgments for packets 3 to 7. They forward these acknowledgments to the *Controller* with the `ack` message, which aggregates them before sending them to the *MC Flow* module with the `aggr. ack` message. The *MC Flow* module del-

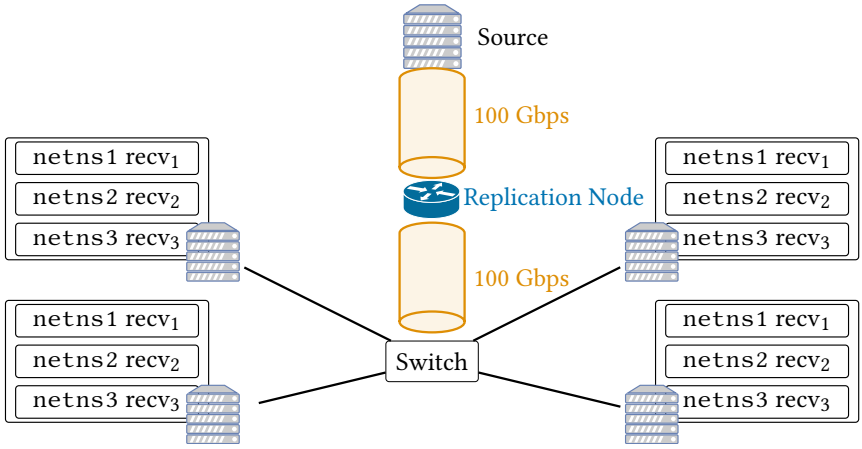


Figure 3.6: CloudLab topology used for our experiments, with three clients per server. We use network namespaces to isolate them. For clarity, the figure only shows 3 clients per server.

egates the frames sent in packets that are considered lost by at least one receiver (`uc_retr(2,5,8)`). The *Controller* dispatches the retransmissions to each unicast path based on the per-receiver acknowledgments.

3.4 Evaluation

We evaluate our implementation on both CloudLab [Dup+19] and emulated networks, respectively, to benchmark its scalability and assess its robustness to failures and changing network conditions in the underlying multicast distribution tree. All our experiments use the Network Performance Framework [Bar24] and are reproducible on CloudLab and commodity servers.

3.4.1 Scalability of FCQUIC

CloudLab topology. We leverage CloudLab [Dup+19] to benchmark Flexicast QUIC. The testbed is illustrated in Figure 3.6. We use 6 *d6515* nodes: the source, the replication node (i.e., emulating the network), and four nodes connected to it via a LAN, 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 up to 1000 receivers.

FastClick router. We emulate the multicast tree on a single node (the replication node) using FastClick [BSM15], which we already used in the previous chapter. Though we do not use HIRT in this evaluation, we leverage the knowledge from previous work. This FastClick router replicates packets

to each receiver using multiple cores to emulate an entire multicast distribution tree. We measured a maximum of 95 Gbps of replicated UDP traffic with payloads of 1200 B. It also forwards unicast packets between the source and the receivers. We prefer this solution over using multiple machines to emulate the multicast network, as resources are scarce on the CloudLab platform. The source code of this software router implementation is also publicly available.

Measured metrics. The source generates traffic at a constant 80 Mbps bit rate, which is considered an upper bound for a 4K video streaming application [24]. Though real video applications would introduce a varying bit rate, we simplify this model by sending 1200 B streams, i.e., the source sends each data block in a different stream.

We measure three metrics while increasing the number of simultaneous receivers:

- *Receiver packet throughput*: the throughput (i.e., including protocol overhead), aggregated on all receivers;
- *Max source CPU*: each second, we poll the CPU usage 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 receivers, accumulated on all receivers.

We compare these metrics with (i) *Baseline*: pure UDP multicast traffic without acknowledgments. Each 1200 B block is sent in a separate UDP datagram, without ensuring reliability. The replication node replicates the packets to the receivers. (ii) *QUIC*: per-receiver unicast QUIC connections. (iii) *FC*: Flexicast QUIC with multicast support; (iv) *FC-5*: Flexicast QUIC with an *ack delay* of 5 ms, again with multicast support; 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 across them to improve scalability. We disable the flow and congestion controllers (both for QUIC and FCQUIC) to push the implementation to its limits.

Unicast QUIC limits. The *Baseline* provides an ideal target cumulative throughput of ~80 Gbps for downward traffic across 1000 receivers. The CPU usage of the source with the Baseline is close to 0 % because the server generates 80 Mbps of unencrypted traffic that is replicated by the network, and does not provide any sort of reliability for the receivers, i.e., there is no feedback nor retransmission mechanism. 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

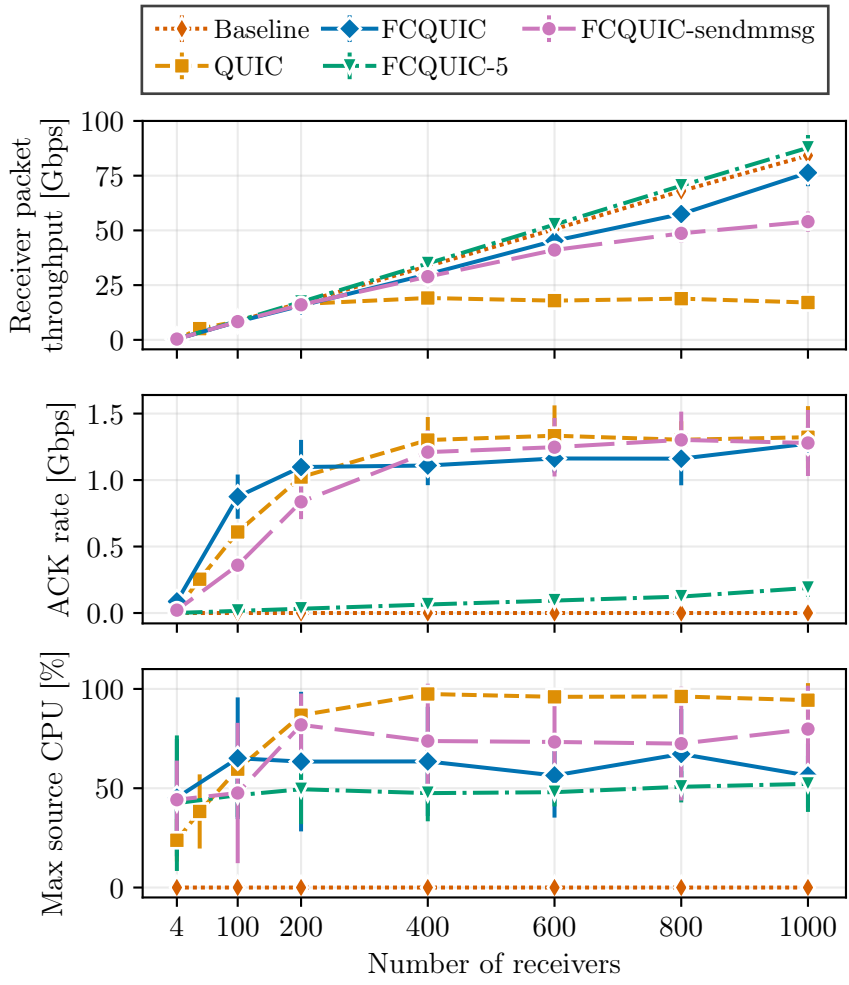


Figure 3.7: Benchmark results using CloudLab [Dup+19]. The source sends 1200 B streams at 80 Mbps. The replication node is emulated using a Fastclick [BSM15] router. We run a UDP Baseline, QUIC, and FCQUIC.

the cap at ~20 Gbps. **More precisely, all 64 available threads reached 100 % utilization after 200 receivers.** As reported in other studies [Kem+24; Tyu+22], the costs of I/O, packet encryption, and connection management are significant, thereby becoming a bottleneck at scale with hundreds of receivers.

FCQUIC supports 1000 receivers and delivers more than 78 Gbps of traffic. Using Flexicast QUIC (the `FCQUIC` curve), the source generates, encrypts, and sends a single copy of each packet on the multicast flow. As such, this process is independent of the number of receivers, and the source

can send data to all recipients as long as the replication node can sustain it.

Acknowledgment rate. Even if the network replicates the packets to the receivers, the source must still handle the acknowledgments from the receivers. A famous problem regarding reliable multicast is the ACK implosion problem [PTK94]. The ACK implosion problem occurs when all receivers send acknowledgments to the source for the same data, creating incast events in the network and potentially overwhelming the source due to packet processing. Because our *Controller* aggregates acknowledgments before sending them to the *MC Flow*, sending new packets on the multicast flow scales to many receivers. Still, increasing the number of receivers significantly increases the load on the source due to these acknowledgments. As such, the maximum CPU usage of FCQUIC stays between 50 % and 75 %. While this value remains acceptable, it shows the impact of accumulating acknowledgments. We also note that the median CPU usage remained under 20 % even for 1000 receivers (not shown in the Figure).

Figure 3.7 further 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 with more than 200 receivers, both QUIC and FCQUIC *ack rates* saturate at ~1.5 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 $\geq$ 400 receivers, we simultaneously run $\geq$ 100 receivers on the same machine. As a result, we observe contention among receivers sharing the same CPU. We noticed that receivers started processing packets in batches when resources became limited, thereby decreasing the *ack rate* because a single PATH_ACK was sent for multiple received packets. As a reminder, QUIC's specification advises acknowledging at least every two ack-eliciting packets [RFC9000] if no *ack delay* is specified [ISK25]. However, our receiver application (both QUIC and FCQUIC) processes I/O reads and writes per batch separately for efficiency, i.e., receivers first read as many packets as possible from the UDP sockets, feed them to the QUIC connection, and then generate the necessary packets on the wire. The asymptotic 1.5 Gbps acknowledgment limit from Figure 3.7 illustrates this contention on the receiving side.

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 of multicast traffic 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 multicast flow. This limit explains the gap between the `Baseline` and FC curve for $\geq$ 600 receivers at the top of Figure 3.7. Without these limits, we expect FCQUIC to scale to 1000 receivers.

Adding an ack delay. We run the same benchmark with an *ack delay* [ISK25] of 5 ms. The source advertises this ack delay in the FC_ANNOUNCE frame. For 1000 receivers, the cumulative *ack rate* remains below 200 Mbps, far below the value without any ack delay. Because the ack delay reduces the number of packets the receivers generate, the source must also generate fewer packets on the unicast paths. As such, the cumulative throughput aligns with baseline expectations and allows FCQUIC to reach ~80 Gbps, similarly to our UDP Baseline. We even notice that the throughput with FCQUIC-5 is higher than the *Baseline*, which is expected due to the byte overhead of QUIC. Though not evaluated, we would expect the cumulative throughput of (unicast) QUIC to also improve with a non-null acknowledgment delay. As such, a fair comparison of the benefits of flexible multicast over unicast should only include the FCQUIC and QUIC curves.

Using *sendmmsg* in non-multicast networks. As explained in Section 3.1, 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 multicast flow. To distribute these packets, the source can *replicate* and forward them as unicast packets. Doing so reduces the source's CPU usage, 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. This system call takes as input *messages* (i.e., a sequence of packets) and a list of destination addresses. For each address in this list, it internally calls *sendmsg*, efficiently distributing multiple UDP packets to that address. As a result, a single user-space system call can send multiple UDP packets to multiple recipients. These `sendmmsg` calls are performed by dedicated threads. The FCQUIC-`sendmmsg` curve on Figure 3.7 shows that by using `sendmmsg`, our FCQUIC source supports many more receivers than QUIC, with a much-reduced CPU load. The cumulative throughput for FCQUIC-`sendmmsg` reaches more than 50 Gbps, more than twice that of QUIC. The maximum CPU usage oscillates between 50 % and 90 %, while the median CPU usage remains below 50 % even for 1000 receivers. We cannot achieve the same throughput as FCQUIC in a multicast-capable network, since replicating packets at the source still incurs higher overhead than using a real multicast tree.

3.4.2 Robustness of Flexicast QUIC

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 3.8 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

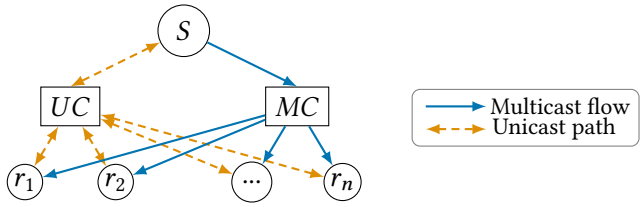


Figure 3.8: Topology used to assess the robustness of Flexicast QUIC. We emulate 20 receivers. *UC* and *MC* are two routers that connect to the receivers in a full-mesh topology.

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 multicast flow.

Video stream. We use FFmpeg [FFMPEG] and GStreamer [GSTREAMER] to generate the video stream and display the video to the user. The video stream is a 180 s replay generated by Baltaci *et al.* [Bal+22]. The video is encoded using the H.264 codec [Wie+03] with a constant rate of 5 Mbps and low-latency parameters. The source uses FFmpeg to generate RTP packets [RFC3550] of ~1200 B. The Flexicast QUIC source sends each RTP packet in a separate stream, meaning that the video is sent reliably to the receivers. The receivers run GStreamer to consume the video stream. We compute the Structural Similarity (SSIM) [Wan+04]. This state-of-the-art metric measures the similarity between each video frame received by the receiver and the frame generated by the source, on a scale of 0 to 1. The higher the value, the more similar the compared frames are; a value of 1 indicates that both frames are identical. We do this computation for each receiver and for each video stream generated by the FFmpeg source. We also report the *frame latency*, i.e., the time it takes for each frame from the FFmpeg source to the GStreamer sinks. Both metrics are important for assessing the quality of experience in video streaming applications.

Failure scenarios. From an operational viewpoint, IP Multicast is more fragile than IP unicast. To enable IP Multicast in a network, operators must enable it on each router's network interface. 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. As a result, 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 %. In other words, every 5 seconds we have a 50 % chance of disabling a new multicast link in the network for 15 seconds. Multiple links can be disabled simultaneously and the same link can be disabled multiple times during the same experiment. We use a seed to reproduce the failure pattern across experiments. We do not notify the FCQUIC source of link failures; instead, we design a scheduler checking for the liveness of the multicast flow with each receiver.

Liveness of the multicast flow. We implement a path scheduler on the source to detect failures in the multicast flow for each receiver. Whenever there are bytes in flight on the multicast flow, the scheduler verifies that it gets *new* acknowledgments from the receiver for data sent on the flow within the *fallback 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 multicast flow is failing, and the FCQUIC source falls back 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 PATH_ACK frames for (new) data received on the multicast flow, the scheduler considers it back alive, and the receiver rejoins the multicast flow. The objective of our scheduler is to provide *seamless* transition between unicast and multicast whenever some link is failing due to temporary failure. We implement this scheduler at the source rather than the receivers to give the source full control over the management of the multicast flow. Indeed, misbehaving receivers may repeatedly leave and rejoin the multicast flow, increasing the load on the FCQUIC server.

This fallback strategy is complementary to the reliability mechanism of (FC)QUIC. As discussed in section 1.5.6, (FC)QUIC marks packets as lost when seeing gaps in the sequence of acknowledged packet numbers. When the multicast link fails, no packets from the multicast flow can be acknowledged, preventing the FCQUIC source from marking gaps for the affected receiver and ultimately avoiding retransmissions. When the receiver falls back on unicast, all in-flight data is retransmitted to the unicast path. If the link failure was significantly shorter than 15 s, the source may detect gaps and retransmit while our path scheduler does the unicast fallback. Both mechanisms could push the same retransmissions through the unicast path, which is handled naturally by the QUIC stack, e.g., because stream data is identified by its identifier and offset, thus avoiding multiple retransmissions.

We evaluate values of the fallback delay (D_{fb}) and the GStreamer playback buffer (G_{pb}) in $\{50, 100, 150\}$ ms. The GStreamer *playback buffer*, G_{pb}, represents the amount of time received video frames are buffered by the application before they are needed (e.g., displayed on the screen). This mechanism adds latency because video frames are not displayed as soon as they

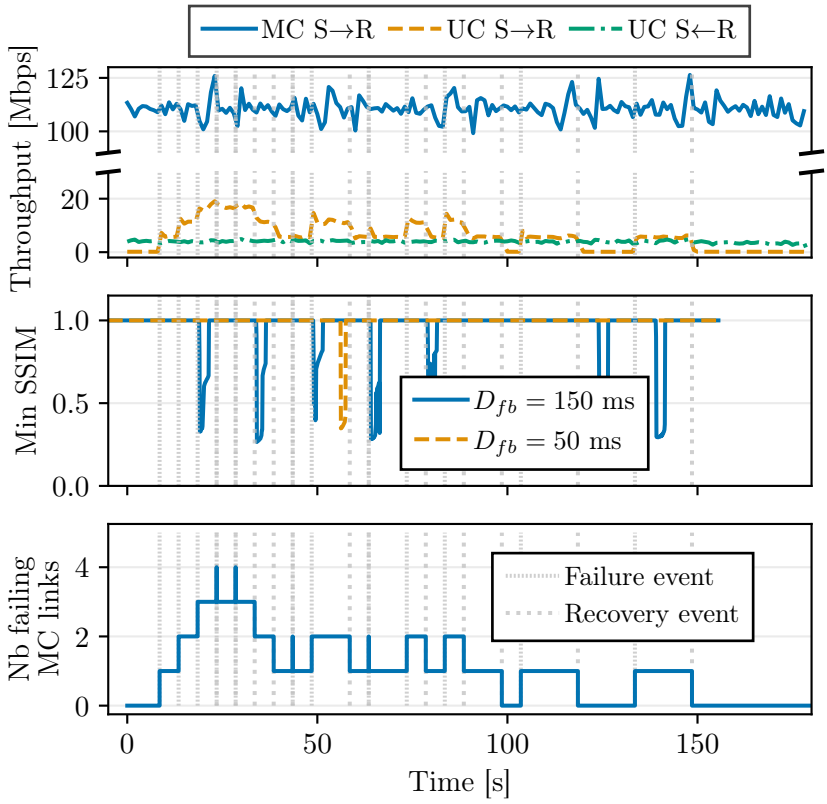


Figure 3.9: Evolution of the throughput, *minimum* SSIM among all receivers for each frame, and the number of failing multicast links for G_{pb} = 100 ms and D_{fb} = 150 ms. Each vertical bar represents a failure or recovery event.

are received, but it reduces the impact of network fluctuations, e.g., transient jitter or packet losses. Common values range around 150 ms [ZOOM].

Figure 3.9 shows the source rates with G_{pb} = 100 ms and D_{fb} = 150 ms. We measure the cumulative bit rate (i) on the multicast flow (MC S→R); (ii) from the source to the receivers on the unicast paths (UC S→R) and finally (iii) from the receivers to the source on the unicast paths (UC S←R). The vertical lines mark the failure/recovery events of the multicast links. The middle graph shows the *minimum* SSIM among all receivers for each video frame. More precisely, for each video frame, we compute the SSIM of all receivers, and only report the one with the *lowest* value. The bottom graph shows the number of failing links at any moment during the experiment. When the scheduler timer is lower than the GStreamer playback buffer ($D_{fb} < G_{pb}$), we expect FCQUIC to detect the link failure and retransmit the lost frames on unicast *before* the GStreamer needs the video frame, and the opposite when $D_{fb} > G_{pb}$.

The Flexicast QUIC source correctly falls back when it detects that

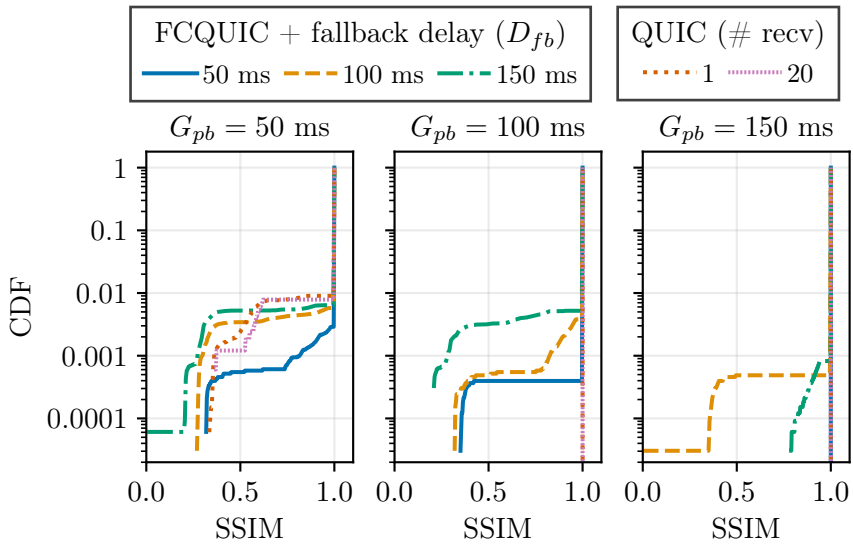


Figure 3.10: Structural Similarity (SSIM) aggregated on all receivers for $G_{pb} = \{50, 100, 150\}$ ms and $D_{fb} = \{50, 100, 150\}$ ms. No link is disabled for the unicast QUIC experiments.

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 a failure for such a receiver, it no longer considers it in the multicast flow. As such, this receiver is not considered when selecting the slowest member for the multicast flow's congestion window. This results in a steady bit rate on the multicast flow for other receivers, despite this receiver's failure. We noticed no significant difference in the throughput when $D_{fb} = 50$ ms.

The fallback mechanism keeps video stream quality even if the multicast tree is failing. Figure 3.10 shows the Structural 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 for ~99 % of the time, indicating that multicast-failing receivers do not significantly impact the others. As expected, the difference between the fallback delay (D_{fb}) and the GStreamer playback buffer (G_{pb}) affects 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 3.10 also reports the SSIM for unicast QUIC, both for 1 and 20 receivers, as a baseline. Because all QUIC traffic passes through the UC router

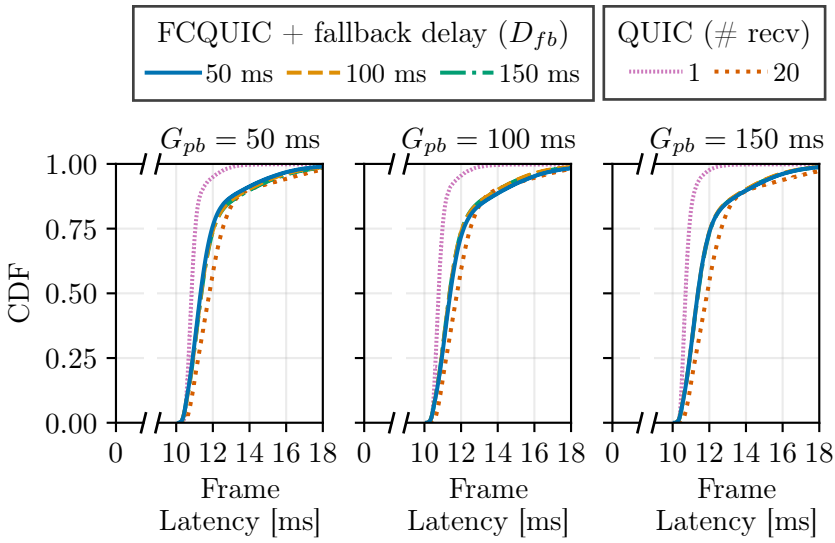
in our topology, the unicast case is unaffected by multicast link failures. Disabling unicast links in this case would just break the connection and stop the video stream. The case with a single receiver shows an ideal baseline, whereas with 20 receivers (i.e., as many as in the experiments with Flexicast QUIC), we see the impact of concurrent requests relative to the unicast case. Except when the GStreamer playback buffer is set to 50 ms, QUIC achieves perfect SSIM across all receivers, as there are no link failures during each run. When G_{pb} = 50 ms, we observe some imperfect frames because QUIC does not always manage to deliver the streams through unicast to all receivers within the small playback buffer with an RTT of ~20 ms.

Figure 3.11 reports the frame latency for the same experiments, aggregated on all receivers – Figure 3.11a shows the global distribution, while Figure 3.11b focuses on the tail distribution, because the link failures impact a small fraction of the video frames.

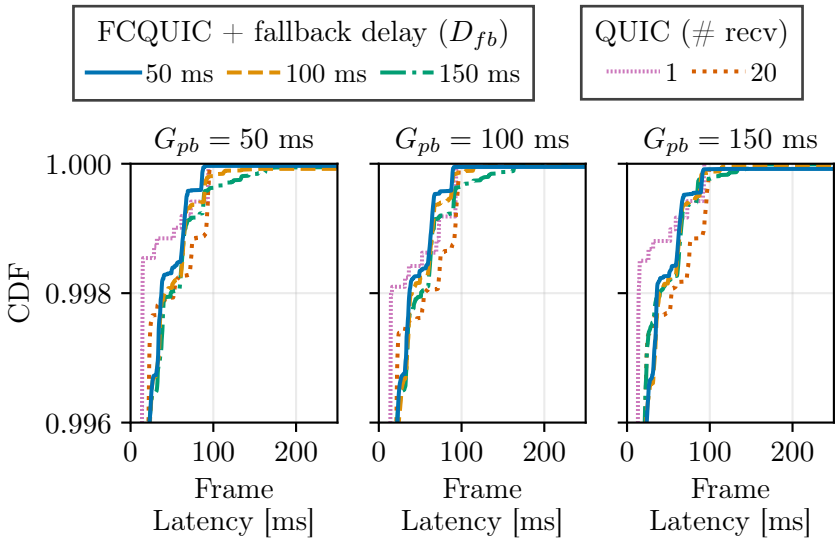
For 20 receivers, Flexicast QUIC slightly improves the frame latency compared to (unicast) QUIC despite the multicast link failures. We notice that the ideal baseline with a single unicast QUIC receiver consistently outperforms FCQUIC and QUIC with 20 receivers. Despite the scalability benefits of using multicast, this is due to the processing of unicast path packets of the 20 receivers. However, the latency provided by FCQUIC remains lower (i.e., better) than QUIC with 20 receivers, because the QUIC server must manage multiple connections in parallel and send data to each receiver individually.

Lower fallback delay improves the tail frame latency. While the latency remains close to the 10 ms one-way delay at the median, 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 [ZOOM]. Finally, we observe that QUIC with a single receiver provides lower tail latency than FCQUIC; QUIC with 20 receivers leads to a higher tail frame latency distribution.

Fallback and recovery latency. Figure 3.12 measures the true latency to detect multicast link failure (left) and recovery (right), depending on D_{fb}. We observe that the proper time to detect that a link is failing depends on D_{fb} and the RTT, as expected, since the scheduler reacts to the *lack of acknowledgment* from the receiver for data sent on the multicast flow. However, the time to consider the multicast flow back alive depends on the RTT, since it takes up to one RTT for the receiver to acknowledge the data and allow the flow to be considered again.



(a) The fallback delay does not significantly impact the frame latency since the failure events are uncommon.



(b) Focus on the last percentile distribution. When $D_{fb} > G_{pb}$, the frame latency increases.

Figure 3.11: Frame latency aggregated on all receivers for $G_{pb} = \{50, 100, 150\}$ ms and $D_{fb} = \{50, 100, 150\}$ ms.

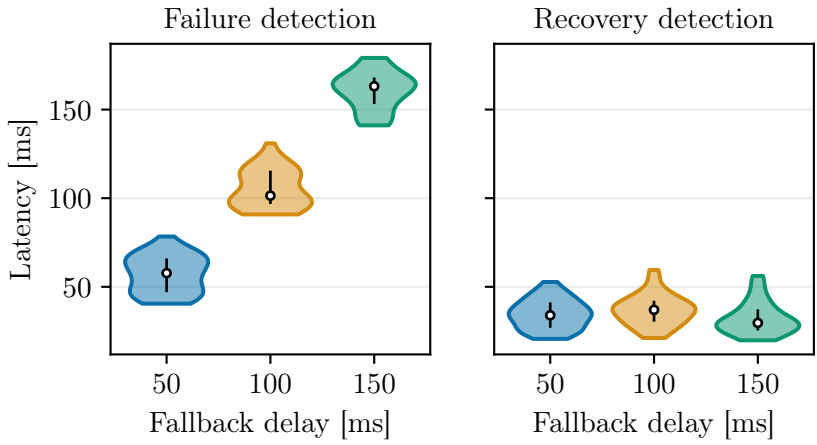


Figure 3.12: Latency of multicast link failure (left) and recovery detection (right) for $D_{fb} = \{50, 100, 150\}$ ms.

3.4.3 Towards managing bottleneck receivers

We now consider the case of heterogeneous receivers, showing that Flexicast QUIC's seamless fallback mechanism ensures good throughput for the multicast flow while handling bottleneck receivers along their unicast paths. We design another path-management algorithm that tracks multicast flow statistics for each receiver, leveraging the fact that the Flexicast QUIC source maintains per-receiver state. Algorithm 2 presents the process running on the source. Every second, the source computes the variance of the round-trip time for each receiver r in the multicast flow, $\sigma^2_{RTT,r}$. If the value is significantly higher than the mean RTT variance computed over the remaining group, the source removes the bottleneck receiver and falls back on unicast.

In parallel, the source checks whether a fallen-back receiver can reintegrate the multicast flow. Such a decision occurs if the two following conditions hold: (i) the RTT variance of the receiver on its unicast path matches that of the remaining members of the group, and (ii) the receiver stayed out of the group long enough. The first condition ensures that the receiver has good enough network conditions to avoid becoming the new bottleneck of the group. We use the RTT variance of the unicast path because the receiver was removed from the multicast flow – thus, it cannot collect new samples on that flow. The objective of the second condition is to avoid oscillations between fallback and re-join of the same receiver. Algorithm 2 uses an initial re-join timer of 5 s, which is increased each time the same receiver falls back, i.e., 5 s the first time, 10 s the second time, 15 s, ... Future work could consider methods to reset this timer if the condition of the bottleneck improves over

Algorithm 2: Bottleneck detection and reintegration mechanism.

γ: RTT variance threshold factor
T_0: initial re-join timer. Init: 5 s
k_r: number of times receiver r fell back. Init: 0
t_r: time since receiver r fell back. Init: None
$\sigma^2_{RTT,r}$: RTT variance of receiver r
$\mathcal{G}$: set of receivers in the multicast group
// Every second, run on source
foreach $r \in \mathcal{G}$ **do**
    $\mu_{-r} \leftarrow \frac{1}{|\mathcal{G}|-1} \sum_{r' \in \mathcal{G} \setminus \{r\}} \sigma^2_{RTT,r'}$;
    if $\sigma^2_{RTT,r} > \gamma \times \mu_{-r}$ **then**
        Remove r from $\mathcal{G}$;
        Fall back r to unicast;
        $k_r \leftarrow k_r + 1$;
        $t_r \leftarrow 0$;
// Check reintegration for fallen-back receivers
foreach $r \notin \mathcal{G}$ ***and*** $t_r \neq None$ **do**
    $\mu_{\mathcal{G}} \leftarrow \frac{1}{|\mathcal{G}|} \sum_{r' \in \mathcal{G}} \sigma^2_{RTT,r'}$;
    if $\sigma^2_{RTT,r} \leq \gamma \times \mu_{\mathcal{G}}$ ***and*** $t_r \geq k_r \times T_0$ **then**
        Add r to $\mathcal{G}$;
        $t_r \leftarrow$ None;

time. Algorithm 2 does not include a hysteresis mechanism after the reintegration of a receiver in the multicast flow, i.e., such a receiver may fall back shortly after rejoining the flow. We do not delay the removal of a bottleneck receiver from the multicast flow to prevent degradation of the quality for the other members of the group.

The Flexicast QUIC source notifies the receiver of whether it is removed and reintegrated from the multicast flow using the FC_STATE frame. Upon reception of this frame, the receiver stops sending IGMP/MLD updates (or, in the case of IGMPv3/MLDv2, explicitly notifies its CPE that it leaves the underlying multicast group). While the client may close the multicast flow using Multipath QUIC's semantics [Liu+26], our implementation, for simplicity, keeps the QUIC path open even if it is no longer used.

RTT variance to fall back. We use the RTT variance as the decision factor, as it is provided by the transport layer. While raw RTT estimates are not sufficient, as distant receivers may provide enough network capacity, the RTT variance captures its evolution and can reveal evolving network conditions. We define a *significantly higher* RTT variance by computing the mean of the group without the potential bottleneck receiver. If the RTT variance of

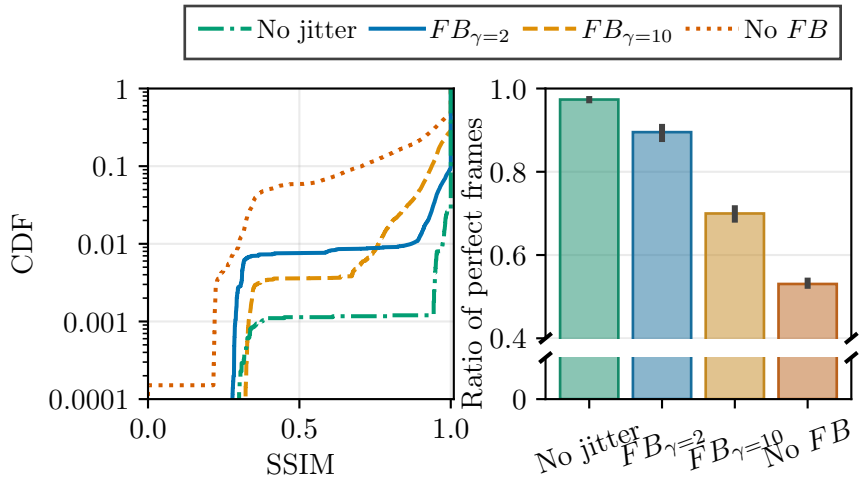


Figure 3.13: Structural Similarity of the 20 receivers when adding 100 ms **jitter and** 1 % **losses and using our bottleneck detection mechanism; 100,000 samples per receiver per experiment. Left: global distribution; right: ratio of perfect frames, i.e., with an SSIM of 1. The GStreamer playback buffer is set to** 100 ms**.**

the receiver is $\gamma\times$ higher than this mean value, we consider it a bottleneck.

Heterogeneous scenarios. We reuse the same testbed as in the previous section to show how an FCQUIC source can handle bottleneck receivers. Instead of entirely disabling multicast links as we did in the last section, we now dynamically add 100 ms jitter (recalling the initial $\sim$ 20 ms RTT) and 1 % losses, using the same method to randomly select links. A selected link is degraded with a probability of 1 for 15 s, every 30 s. We run each scenario 3 times.

The mechanism ensures a good video quality. Figure 3.13 reports the Structural Similarity (SSIM) in this scenario. The *No jitter* acts as a baseline; the three other curves show the addition of jitter and losses. Without our bottleneck and fallback detection mechanism, the SSIM degrades, with 1 % of frames being of low quality due to jitter and packet loss. Figure 3.13 also reports the SSIM when using the bottleneck fallback mechanism with $\gamma = \{2, 10\}$. Both values significantly improve SSIM (i.e., video quality) because heterogeneous receivers are evicted from the multicast flow and continue to receive traffic via their unicast paths, which are not affected by jitter or loss. Still, the mechanism cannot guarantee the same SSIM as the baseline due to the time required to detect bottleneck receivers. The right-hand side of the Figure shows the proportion of perfect video frames (i.e., with an SSIM of 1), showing substantial gains with $\gamma = 2$ over $\gamma = 10$ because the

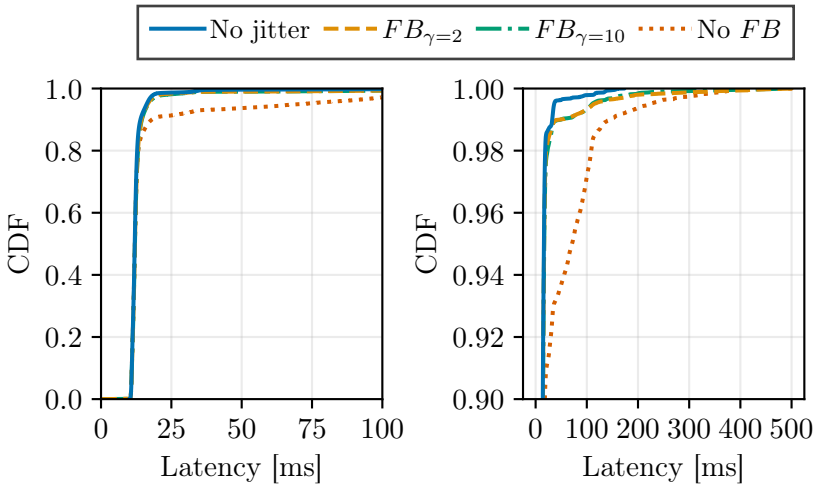


Figure 3.14: Frame latency of the 20 receivers when adding 100 ms jitter and 1 % losses and using our bottleneck detection mechanism; 100,000 samples per receiver per experiment. Left: global distribution; right: focus on the tail latency.

bottleneck detection system is more aggressive. We again notice the gains of the mechanism compared to the *No FB* curve: from 50 % to > 75 % of perfect frames.

The mechanism keeps low frame latency. Figure 3.14 shows the frame latency in the same experiments. Again, we observe the impact of the jitter and losses on the whole group without our detection mechanism, as almost 10 % of the frames experience > 3× latency. At first glance, the gains both in SSIM and frame latency could stem from the unicast fallback, where we do not degrade the link. A deeper look at the raw results shows that both metrics worsen for all members of the multicast flow when link degradation occurs, as the source adapts its transmission rate to the bottleneck receiver, whose throughput is reduced by the observed losses and jitter. Similar to SSIM, frame latency improves with the mechanism, though we do not observe a difference with γ.

The source may wrongly reintegrate receivers in the multicast flow. Figure 3.15 shows the number of times we degrade the quality of a multicast link, and how the fallback mechanism reacts to them. Lower γ leads to more aggressive fallback decisions, as we see in the Figure. We also note that for both $\gamma = 2$ and $\gamma = 10$, the number of fallbacks exceeds the number of link degradations. Let us recall that we degrade links for 15 s, while the FCQUIC source checks every second for potential bottleneck receivers. The source also tries to reintegrate such receivers if their link quality improves over time, ini-

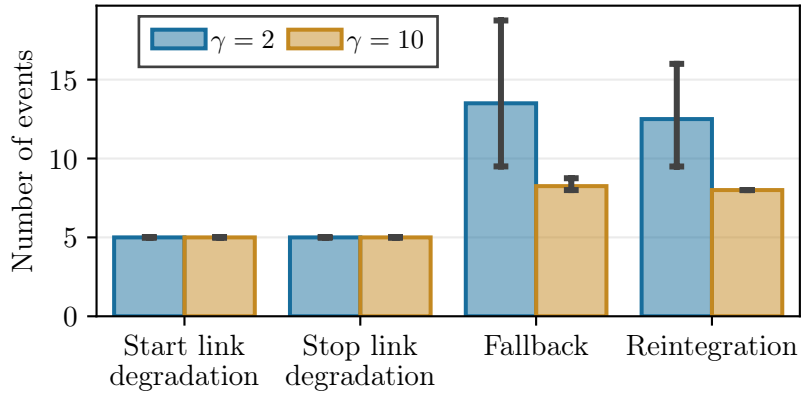


Figure 3.15: Number of link degradations and unicast fallback as decided by our bottleneck detection mechanism.

tially after 5 s. In our experimental setup, the unicast path is not affected by jitter/losses; thus, the fallback receiver reports good network path quality, allowing the source to reintegrate it into the multicast flow. In that regard, the source may reintegrate such a receiver even while the multicast flow's network conditions remain degraded. The FCQUIC source then reconsiders this receiver as a bottleneck, approximately one second later.

To avoid such an oscillation every five seconds, Algorithm 2 includes a hysteresis, where the wait time to reintegrate the multicast flow depends on the number of times the receiver has already fallen back. Figure 3.16 shows the time receivers remained out of the multicast flow, which linearly increases as the same receiver experiences multiple link degradations or because the link is still degraded once it reenters the multicast flow. From the results discussed above, $\gamma = 10$ allows the Flexicast QUIC source to detect bottleneck receivers without generating too many oscillations compared to $\gamma = 2$, though this is specific to the jitter being 10× higher than the one-way delay we applied in this section.

This observation is further illustrated by Figure 3.17, which shows the unicast-to-multicast throughput ratio as measured each second on the Flexicast QUIC source. With $\gamma = 10$, the source sends the data *at most* once via unicast, i.e., there is a single fallen-back receiver at a time – recalling that we degrade multicast links one at a time. With $\gamma = 2$, the bottleneck detection mechanism is more aggressive, resulting in multiple receivers being removed from the multicast flow *simultaneously*, indicating that more unicast fallbacks occur than expected by our degradation experiment. This is confirmed by a closer look at our results, which indicates that with $\gamma = 2$, the FCQUIC source sometimes removes non-impacted receivers from the multicast flow, e.g., be-

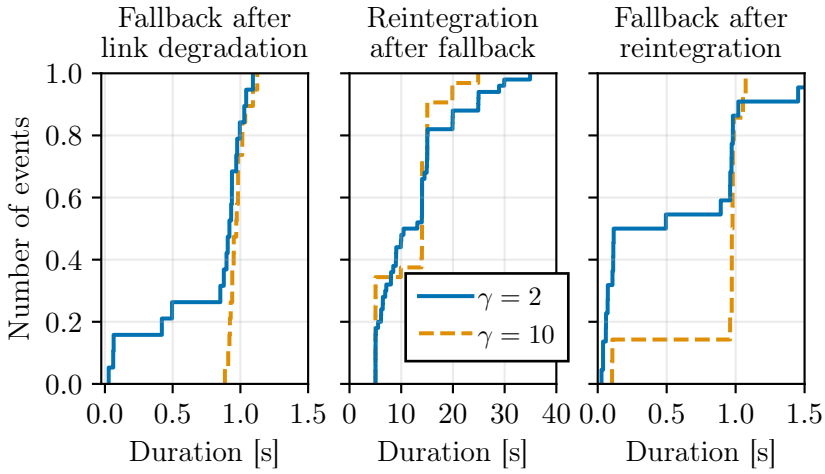


Figure 3.16: From left to right: time to detect a new bottleneck receiver due to link degradation, to reintegrate a fallen back receiver, to reconsider a reintegrated receiver as a bottleneck because the link is still degraded.

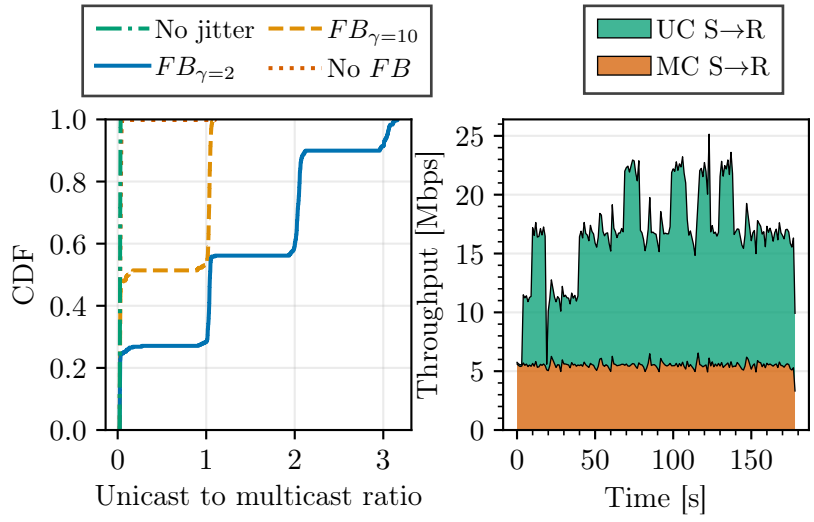


Figure 3.17: Left: unicast to multicast throughput ratio as measured each second on the FCQUIC source. Right: throughput of the FCQUIC in a specific experiment with our mechanism $\gamma = 2$.

cause their RTT variance (0.52 ms) is more than twice that of the rest of the group (0.217 ms), resulting in a false positive. This ultimately results in higher unicast traffic, as Figure 3.17 shows, with up to three times the multicast rate sent to unicast receivers.

Discussion. In this section, we derive a simple bottleneck receiver detection mechanism based solely on the RTT variance. Future work must consider more advanced decision processes, e.g., accounting for the observed losses. Though simple, this mechanism shows how the source handles bottleneck clients and that the extension discussed in this chapter contains all the necessary features for this purpose. These preliminary results show that a too-aggressive decision process (e.g., $\gamma = 2$) leads to false-positive unicast fallbacks, affecting network efficiency. Next steps should involve measurements in real networks, where bottleneck receivers may expose different behaviors.

3.5 Related work

This chapter focuses on the design and evaluation of Flexicast QUIC, a flexible approach for providing multicast at the transport layer. We focus on works related to Flexicast QUIC at the transport layer.

Two IETF drafts have proposed to extend QUIC to support multicast. HTTP over multicast QUIC [PBH22] suggests using HTTP/3 and 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 been implemented only partially in nghq [NGHQ] and has not been further explored for several years.

Multicast extensions for QUIC [Hol+26] specify extensions to enable QUIC to use a multicast distribution tree. These extensions focus on source authentication to prevent spoofing by malicious clients. It suggests using the AMBI [HRF25] method, which forwards multicast packet hashes either via a Merkle tree on multicast or via per-client unicast connections. In a companion paper [FHS24], the authors present MCQUIC, a deployment architecture that uses fan-out servers to handle unicast connections to each client and a side one-to-many connection to distribute data to receivers. There is an open-source implementation of MCQUIC [MCQUIC-IMPLEM] 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 [Hol+26]. FCQUIC leverages and extends Multipath QUIC, simplifying the design and enabling seamless fallback to unicast when multicast trees fail. In contrast, MCQUIC extends QUIC but does not specify methods for handling transport-related features such as reliability and congestion control. We benchmark Flexicast QUIC at high

speed and present alternatives (e.g., `sendmmsg`) when no underlying multicast tree exists. We focus instead on scalability, robustness, and evaluations of our implementation.

MCQUIC [Hol+26; FHS24] focuses on the source authentication problem, which we do not investigate in this chapter. We previously investigated the cost of source authentication in a technical document describing the first version of Flexicast QUIC [NPB23]. There, we observed the impact of using asymmetric signatures to authenticate multicast packets, which are orders of magnitude more costly. On the other hand, AMBI [HRF25] uses efficient packet hashes to authenticate packets at the cost of sending these hashes using unicast to each receiver, which breaks the spirit of multicast. To gain efficiency, AMBI suggests using Merkle trees, but this mechanism is not loss-tolerant. Another possibility for future work would investigate the ALTA [RH19] method.

3.6 Discussion

This chapter presented Flexicast QUIC (FCQUIC), an extension to QUIC and its multiple paths management addition [Liu+26]. With FCQUIC, we revisited our vision of multicast-capable transport protocols, using multicast *if* possible for efficiency and unicast otherwise for robustness. Thanks to the QUIC extension for multiple paths [Liu+26], FCQUIC exposes per-receiver bidirectional unicast paths and shared unidirectional multicast flows within the same connection, enabling seamless transitions between multicast and unicast delivery for receivers. Except for defining new flexicast-dedicated QUIC frames, we extended QUIC to use a distinct set of TLS keys when opening new paths, which was already considered during discussions of Multipath QUIC [QUIC-ML-2020]; and to prevent receivers from sending any data on the multicast flow, which has already been explored in Multiflow QUIC [DB21]. We presented the design of Flexicast QUIC and provided an open-source implementation based on Cloudflare *quiche* [QUICHE]. We highlighted its scalability, with cumulative traffic exceeding 80 Gbps, and its robustness in scenarios where receivers may lose multicast connectivity or experience degraded network quality, with minimal impact on their video stream quality of experience.

Possible improvements. The evaluation from Section 3.4 aimed to show the benefits of Flexicast QUIC, both in terms of scalability and robustness. First, the benchmarks consider a perfect multicast network without heterogeneous receivers and without packet losses that would impact the maximum cumulative throughput in the presence of retransmissions. Second, our robustness evaluation when adding link failures (section 3.4.2) and heterogeneous receivers (section 3.4.3) considers basic scenarios and presents simple

path management mechanisms to handle these conditions. Though this chapter aims to demonstrate the flexibility of Flexicast QUIC, improvements to this work include more advanced mechanisms and more complex scenarios, possibly in real networks. Finally, these sections introduce multiple parameters (D_{fb} and γ) that are set to fixed values. An improvement would consider how to derive these values dynamically, e.g., based on each receiver's RTT.

Conclusion. This chapter focused on providing a flexible, multicast-capable transport protocol, assuming at least partial multicast connectivity from the source to the receivers. However, at the time of writing, such end-to-end multicast connectivity is too much of an assumption to imagine real deployment of Flexicast QUIC on the Internet. We acknowledge that this chapter raises several questions that we do not address directly here, including those detailed above. Despite this, we believe that Flexicast QUIC paves the way to answering several open research questions regarding multicast in general in the future. The next chapter focuses on one of them: the *ACK implosion* problem inherent to multicast communication.

Scaling Flexicast QUIC for Large-Scale Software Distribution

4

Chapter 3 introduced Flexicast QUIC as the possibility to combine multicast and unicast within the same transport connection, highlighting its design and the changes to QUIC and its multiple paths management extension. We evaluated FCQUIC in the context of constant-rate data, mimicking a simplified high-quality live-streaming application, and outlined the scalability benefits of the proposed approach compared to unicast QUIC. More particularly, we outlined the interest of delaying acknowledgments from FCQUIC receivers to improve scalability since the source must process fewer packets. Still, we stuck to a constant, pre-defined acknowledgment delay.

This chapter more deeply explores the impact of receiver pacing on acknowledgments in Flexicast QUIC communication by reconsidering the *ACK implosion* problem in multicast:

> In reliable multicast, a single data packet triggers independent acknowledgments from all receivers, leading to a large number of packets coming back to the source (i.e., incast) and requiring packet processing that increases linearly with the group size.

Flexicast QUIC maintains per-receiver unicast paths, allowing the server to know at any time the number of active receivers listening to the multicast flow. We leverage this additional knowledge to build an *adaptive acknowledgment delay* strategy, in which the server computes and advertises a delay based on its sending rate, the group size, and the maximum acknowledgment rate it can support. We analyze the benefits and trade-offs of this approach in the context of large-scale file delivery, which puts pressure on Internet Service Providers (ISPs) with each new popular video game release or software update, e.g., on Windows and iOS devices. Since receivers request the same file in concentrated but not simultaneous bursts, we further design a file-rolling system that splits the file into blocks and sends them in a carousel, allowing late receivers to begin processing data at the next block boundary rather than waiting for a full retransmission.

We implement and evaluate both mechanisms in FCQUIC on CloudLab with up to 1000 receivers. We show that the adaptive strategy reduces the acknowledgment rate by an order of magnitude without degrading per-receiver goodput, and compare FCQUIC to unicast QUIC and multicast state-of-the-art protocols across numerous scenarios, including heterogeneous arrival rates and packet losses. We further model the CPU load of the FCQUIC source and show that the acknowledgment rate is its dominant factor, and that it can be controlled using our adaptive acknowledgment mechanism, enabling larger-scale deployments.

The remainder of this chapter is organized as follows. Section 4.1 motivates this chapter, underlining the impact of large-scale software updates on today's ISPs. We derive our *adaptive acknowledgment* strategy in Section 4.2, highlighting the trade-off between scalability and reactivity to loss and congestion events. In Section 4.3, we present the *file-rolling* mechanism, which allows late receivers to start processing the application payload without waiting for the current multicast transmission to start from the beginning. We implement our two systems in Flexicast QUIC and evaluate them in Section 4.4, showing their benefits compared to unicast QUIC and NORM [RFC5740], the state-of-the-art regarding multicast file delivery protocols, and we analyze the CPU load of the FCQUIC server. Finally, we conclude this chapter in Section 4.5.

The results from this chapter are currently under review for a journal.

4.1 Large-scale software updates

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 [Uni25]; new triple-A games usually weigh multiple tens of gigabytes [AJ 25]. 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 [Cla+17]. 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 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 Rome at Namex [Nam25d]. 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, reported that another Fortnite patch pushed traffic to 372 petabytes in a single day [ope25], 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 [Ble+18]. This study also found that the 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 [Nam25c] and Apple [Nam25a] update releases.

In response, ISPs and IXPs must scale their networks to ingest these sporadic events [Nam25b], leading to an order-of-magnitude increase in network capacity beyond daily requirements [Ber25]. 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 [Aka26]. Applications also try new solutions to prevent saturating their customers' network: in early November 2025, the release of the Call of Duty game started with a pre-download phase [Dut24] that enabled users to download most of the game before finalizing it on the release day to prevent traffic spikes. While a file transfer is not a one-to-many application at first glance, these recent events indicate a concentration of requests for duplicate content within short periods [Nam25d; ope25; Ble+18; Nam25a].

A more efficient way to distribute files is to rely on *multicast* protocols. Multiple previous works attempted to disseminate files efficiently using multicast [GWP99; GMP96], leading to standardized protocols [RFC5740; RFC6726]. However, these protocols assume an end-to-end (from the source to all receivers) working multicast network, and do not provide unicast fallback in case subsets of receivers lack such connectivity. As a result, multicast file transfer is confined to enterprise networks [Mic22], where operators have complete control over their networks and their hosts, or specific applications in research networks [Sto+20].

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

the existing intra-domain multicast deployments to efficiently distribute such updates using Flexicast QUIC. File distribution begins with the establishment of a regular QUIC connection and an HTTP/3 request from clients. If the server can distribute the file through efficient multicast, it upgrades the connection to use Flexicast QUIC. The protocol's flexibility allows clients that become a communication bottleneck or lack multicast support to 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 [LG96]. Recent multicast protocols use negative-acknowledgment (NACK)-based approaches to limit the scope of the problem [RFC5740], 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. Based on 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.

4.2 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 [RFC9000]. 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.

Ack implosion [PTK94] occurs when the number of acknowledgments the sender must handle for every transmitted packet increases due to the number of simultaneous receivers. 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 [RFC5740]. By aiming for arbitrarily large scalability, these protocols do not track the number of active multicast receivers, which is necessary to scale to large audiences, and must thus define methods that limit the ack rate independently of the

group size. In this chapter, 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 [RFC1122] and QUIC [ISK25]. TCP endpoints must not delay acknowledgment by more than 500 ms [RFC1122]. The IETF draft on QUIC acknowledgment frequency recommends that endpoints send an ACK frame at least *once per round trip* to minimize potential negative impact on congestion and reliability mechanisms.

4.2.1 Estimating the acknowledgment rate

Chapter 3 explored the benefits of adding an acknowledgment delay in the initial design of FCQUIC, suggesting improved scalability because the source must process fewer packets. 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 4.1 summarizes the parameters we use in this chapter for clarity. 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:

$$r_a(t) \propto r(t - \delta_t) \times n(t - \delta_t). \tag{4.1}$$

The δ_t in Equation (4.1) 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.

Typical networks use an MTU of 1500 bytes. For bulk data transfers, QUIC packets containing data (i.e., STREAM frames) try to maximize this MTU, while packets with 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* [QUICHE]. We derive that

$$r_a(t) \approx \frac{1}{15} r(t) \times n(t) \tag{4.2}$$

4.2.2 Limiting the acknowledgment rate

We define C as the maximum expected acknowledgment rate summed over all receivers. C captures the maximum acknowledgment processing capacity

Symbol	Unit	Description	Page appearance
r	bps	Data rate	109
r_a	bps	Acknowledgment rate	109
n	–	Number of active receivers	109
β	s	Acknowledgment delay	110
C	bps	Acknowledgment cap	110
α	–	Pareto shape	114

Table 4.1: Description of the parameters.

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 least) two ack-eliciting packets [RFC9000], 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 of the same receiver 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:

$$\mathbb{E}[r_a(t)] \approx \frac{800 \times n(t)}{\mathbb{E}[X(t)]} = \frac{1600 \times n(t)}{\beta(t)}, \tag{4.3}$$

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 (4.4).

$$\beta(t) = 1600 \frac{n(t)}{C}. \tag{4.4}$$

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

$$\beta(t) = \begin{cases} 0, & \text{if } \frac{r(t)\times n(t)}{15} \leq C, \\ 1600\frac{n(t)}{C}, & \text{otherwise.} \end{cases} \tag{4.5}$$

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 (4.6):

$$C \geq \frac{1600 \times n_{\max}}{2 \times RTT_{\max}}. \tag{4.6}$$

In this computation, $RTT_{\max}$ corresponds to the highest RTT among all receivers. We use $RTT_{\max}$ to adapt to the receiver with the longest delay, even though other receivers may have much lower RTTs. This $RTT_{\max}$ is estimated from packets sent on multicast flow 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 [RFC9000]. The source accounts for this quantity when adjusting its RTT samples.

Moreover, adding a delay introduces batched acknowledgments, which free space in the sender's congestion window and allow the emitter 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 4.4.3.

Distributing the ack delay. We extend FCQUIC with a new frame, FC_ACK_DELAY, 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 indi-

vidually 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 4.4.2.

Handling heterogeneous receivers. Our previous reasoning, which infers the computation of $\beta(t)$ from Equation (4.5), 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 [ISK25], 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 (4.6)). For simplicity and multicast efficiency, we use a single β and assess the impact in Section 4.4. This represents a deliberate trade-off between ack rate control and congestion-control reactivity, which we leave to future work.

4.3 Enabling loose synchronization through file rolling

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 expected by design. Recent events [Nam25d; ope25; Ble+18; Nam25a] 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 [RFC5740] 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. FLUTE [RFC6726] uses Asynchronous Layered Coding [RFC5775] to transmit encoding symbols over a unidirectional channel, allowing receivers to reconstruct the original object once enough symbols have been collected, without requiring acknowledgments or retransmissions. 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.

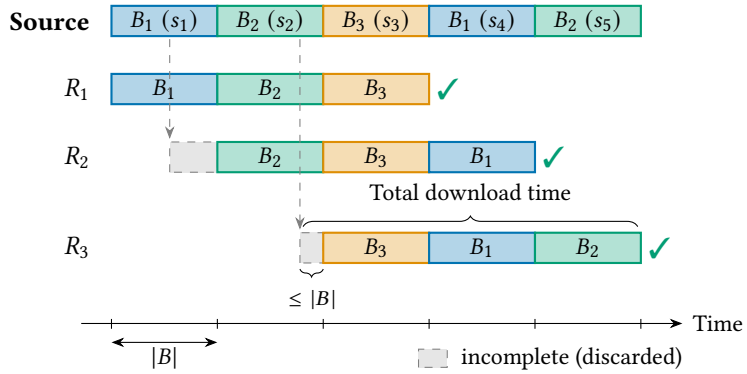


Figure 4.1: File-rolling example with a file split into three blocks (B_1, B_2, B_3). R_1 joins at the start and receives all blocks in order. R_2 and R_3 join mid-block: the incomplete block is discarded, and reception resumes at the next block boundary. Each block is repeated in consecutive QUIC streams ($s_1 \ldots s_5$).

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. Figure 4.1 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 at any time (e.g., receiver R_3 in Figure 4.1). 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 [RFC9000], making stream ID exhaustion a non-concern.

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 Figure 4.1, 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 4.4.5 by implementing the file-rolling system in FCQUIC.

4.4 Experimental Evaluation

In Chapter 3, we provided a first open-source implementation [Nav26b], based on a previous specification of the multiple paths management extension for QUIC [Liu+24]. We update this implementation to build on the latest versions of the IETF draft [Liu+26] 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 [Dup+19]. We compare unicast QUIC with Flexicast QUIC as we deliver the same file to an increasing number of receivers. First, we show the benefits of FCQUIC regarding network utilization (§4.4.1). We assess how our ack delay strategy dynamically caps the acknowledgment rate based on the number of active receivers without compromising the transmission rate (§4.4.2). Then, we infer a model estimating the CPU load of the Flexicast QUIC server to theoretically extrapolate to larger sets of receivers (§4.4.3). 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 [RFC5740].

CloudLab setup. We leverage CloudLab [Dup+19] to run a medium-scale multicast network, reusing the same testbed as in Chapter 3 (Figure 3.6). 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 similarly to the previous chapter. 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 cumulative replicated multicast traffic is capped at the 100 Gbps link speed.

Receiver arrival behavior. We model the client's arrival behavior using a *Pareto distribution*, a power-law model for event bursts with long tails, corresponding to the recent software dissemination events [Nam25d; ope25; Ble+18; Nam25a]. 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 4.7:

$$P(X > x) \sim (x_m/x)^{\alpha}, x \geq x_m \tag{4.7}$$

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 more than 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 [QUICHE]), NORM (version 1.5.9 [Lab25]), which we introduced in section 1.8.3, and our new implementation of FCQUIC. For FCQUIC, we analyze three variants: (i) FCQUIC: using the standard QUIC acknowledgment mechanism; (ii) FCQUIC$_{C,5}$: using a constant *ack delay* of 5 ms (similar to Chapter 3); (iii) FCQUIC$_A$: using our adaptive acknowledgment strategy from Section 4.2. 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 to fairly compare the two multicast protocols. We only implement our file-rolling system in FCQUIC, though a similar mechanism could be built atop NORM. As such, we expect FCQUIC to perform better than NORM for large sets of clients, as FCQUIC receivers must only wait for the next block, compared to NORM, which requires waiting for the current file transmission to complete.

4.4.1 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. Figure 4.2 shows the per-receiver goodput and Figure 4.3 the throughput measured on the Replication Node. Except stated otherwise, we use a maximum acknowledgment rate cap of $C = 100$ Mbps and a block size $B = 100$ MB. We analyze the impact of these two parameters in Sections 4.4.2 and 4.4.5.

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 [Jae+23]. This value decreases linearly with the number of receivers, reaching a median of less than 200 Mbps at 200 receivers.

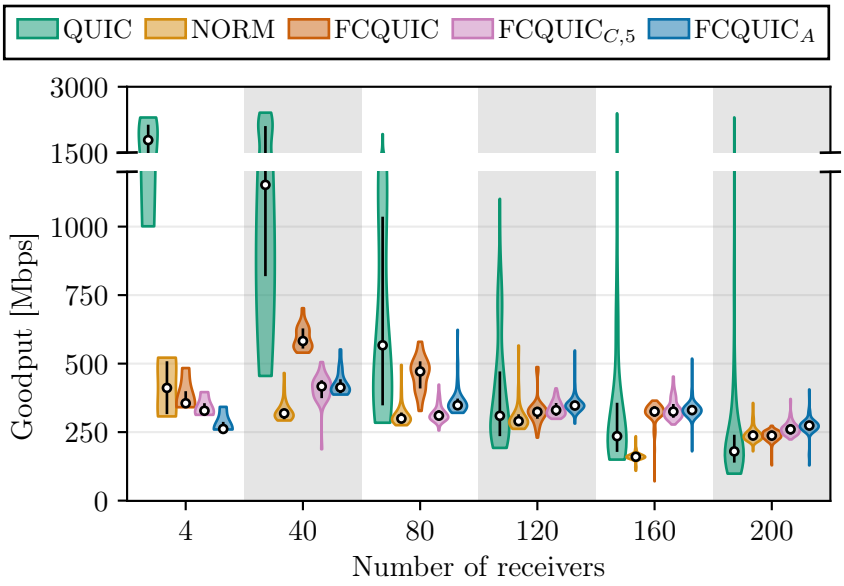


Figure 4.2: Application goodput in our CloudLab setup. FCQUIC outperforms (unicast) QUIC with numerous receivers. Dots show the median value.

The measured unicast throughput (including protocol overhead) in Figure 4.3 indicates that the server reaches a peak of 30 Gbps. Similarly to the results from Chapter 3, the server CPU load reaches full utilization for QUIC, which becomes the bottleneck before the bandwidth limit. Again, we observed CPU contention on the nodes 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. Figure 4.3 shows that FCQUIC linearly increases the cumulative multicast throughput until it reaches ~100 Gbps for 200 receivers, explaining the better results compared to QUIC.

In a perfect environment, NORM provides similar goodput to FCQUIC. Except for 4 receivers, FCQUIC variants 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 – because these retransmissions occur on the multicast channel. The link utilization of NORM is systematically

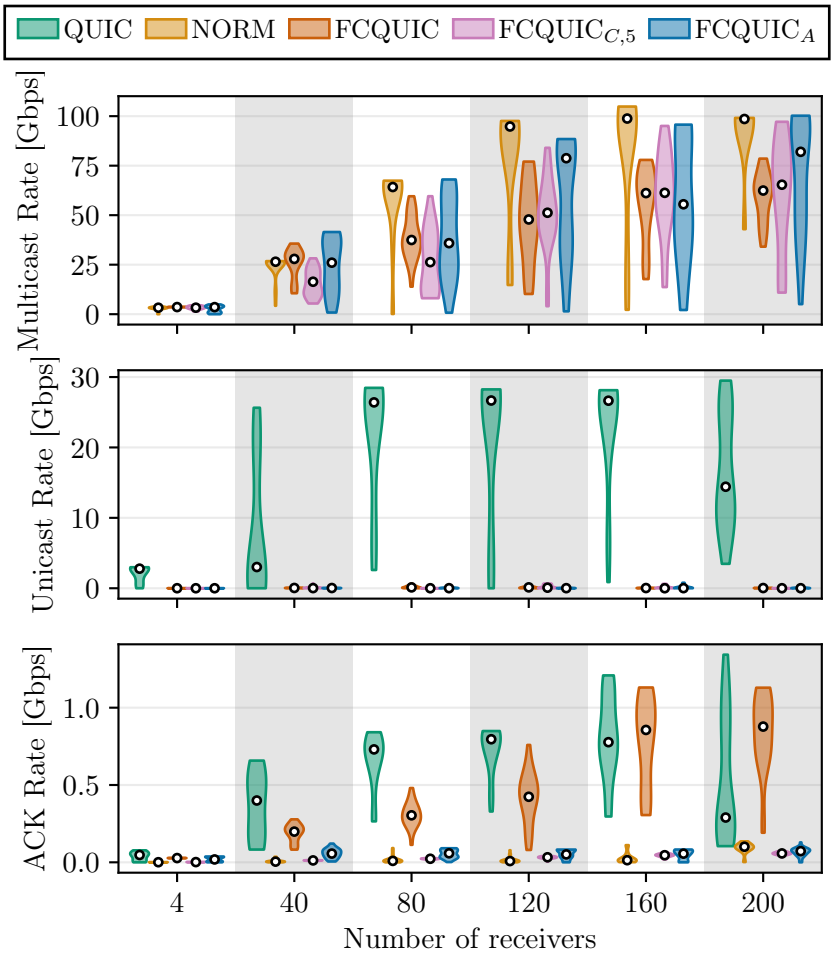


Figure 4.3: Throughput measured on the Replication Node during the experiments from Section 4.4.1: Multicast (top), unicast (middle), and ack (bottom) rates.

higher than that of FCQUIC, despite the lower per-receiver goodput. We observed significantly higher drops from the software router with FCQUIC than with NORM: for 200 receivers, the router dropped 14, 974 FCQUIC$_A$ 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 goodput difference is not significant, both figures show a positive impact of the adaptive strategy (FCQUIC$_A$, using a maximum rate of $C = 100$ Mbps)

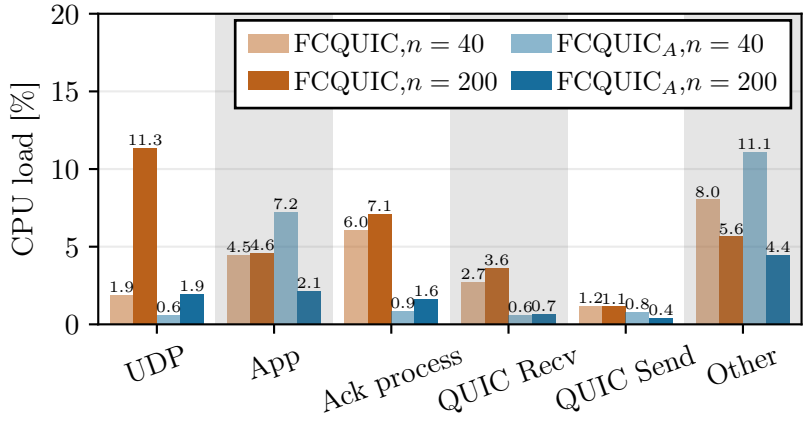


Figure 4.4: Contributors of the CPU load on the FCQUIC source, comparing standard and our adaptive acknowledgment strategy, for 40 and 200 receivers. We only report elements that vary significantly with n. *App* includes application-specific tasks, such as reading data from a disk file. *QUIC Send* typically involves tasks such as sending packets on the multicast flow and per-receiver retransmissions. We used the `perf` tooling suite to collect these samples.

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 cumulative 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 does not adapt to the number of active receivers.

Our adaptive strategy lowers the CPU load on the FCQUIC source. In Figure 4.4, we report the main contributors of the CPU load on the source using the standard and our adaptive acknowledgment strategy, measured using the `perf` toolkit. We exclude 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. Indeed, to improve the scalability of our FCQUIC$_A$ source, we updated the design from Figure 3.4 to let each unicast path instance embed a *connected* UDP socket. This tweak avoids packet copies through message passing, at the cost of creating a new socket for each receiver. With FCQUIC$_A$, the kernel polls these sockets less frequently, reducing the CPU load from 11.3 % to 1.94 % for $n = 200$ receivers.

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 (i) because the FCQUIC$_A$ source processes fewer packets, tasks are more often in an idle state, and `tokio` must wake them up more regularly; (ii) `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. Our *file-rolling* system allows FCQUIC to outperform 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.

4.4.2 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. Figure 4.5 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 $\beta \approx 30 \times \text{RTT}$.

Such a large value of β yields batched acknowledgments. As discussed in Section 4.2, 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 Figure 4.6 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

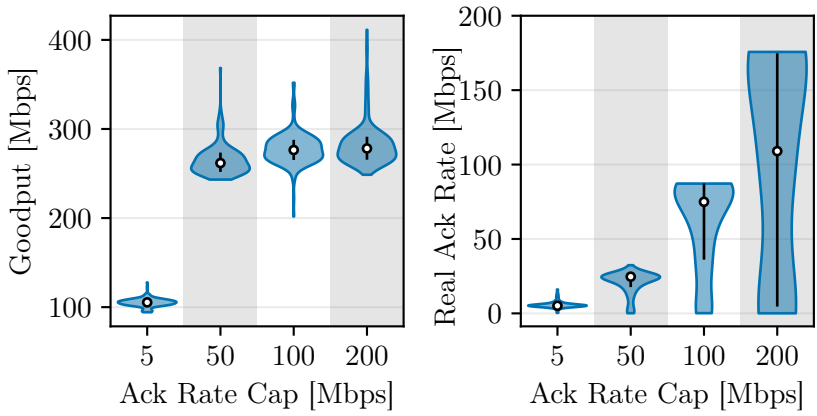


Figure 4.5: Per-receiver goodput and real measured acknowledgment rate (r_a) when varying the acknowledgment rate cap (C), using our adaptive strategy FCQUIC$_A$.

that the Flexicast QUIC source distributes the updated β values through the FC_ACK_DELAY frames every 100 ms. The right-hand side of Figure 4.6 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.

4.4.3 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 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 $n = 1000$ re-

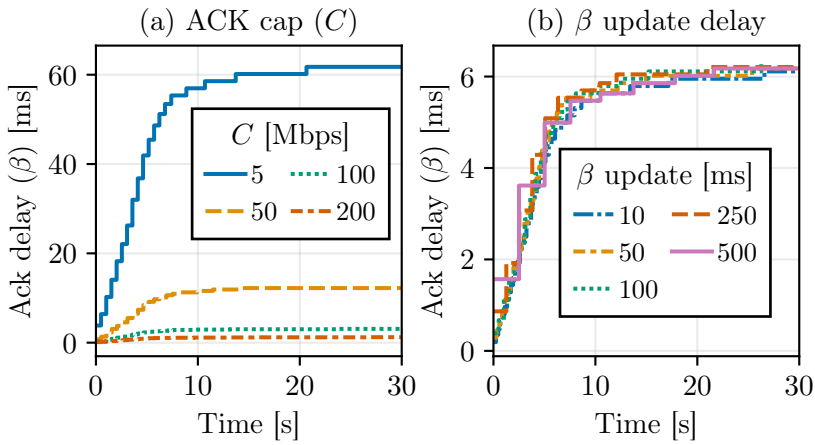


Figure 4.6: The acknowledgment delay β adjusts to the maximum cap C (left); increasing the β update frequency adjusts the acknowledgment rate for receivers more smoothly (right), though the per-receiver goodput did not get affected for the tested ranges of frequency update.

ceivers, in increments of 100 receivers, using FCQUIC$_A$ and multiple acknowledgment rate caps (C). Figure 4.7 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 top 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., comparing curves for the same number of clients) 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 top part of Figure 4.7 also shows that, for larger values of C, the CPU load increases with n, the number of active multicast receivers. Two reasons explain 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 more (and larger) acknowledgments as the number of receivers increases to ensure full reliability. Second, the third sub-figure shows the measured acknowledgment rate during these experiments. For $C \geq 125$ Mbps, the ACK rate continues to increase when $200 \leq n \leq 1000$ because the acknowledgment cap is large

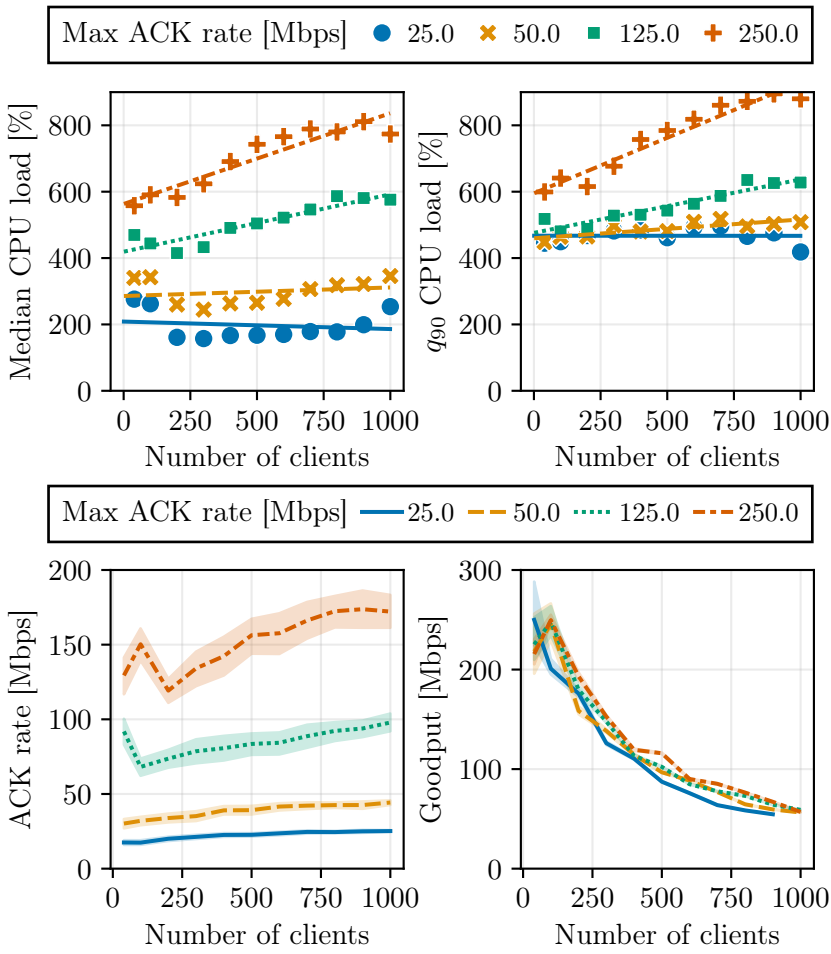


Figure 4.7: Median and q_{90} of the CPU load when increasing the maximum acknowledgment rate (C) and the number of receivers (n) with FCQUIC. Scatters represent real samples from the CloudLab evaluation; the lines are computed using least-squares polynomial fits of order 1. The bottom sub-plots, respectively, 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.**

enough so that $\beta \approx 0$. To some extent, 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 measure the correlation between r_a, n and the CPU load in Table 4.2. The table highlights that the linear dependence between CPU load and the acknowledgment rate is strong, with a

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 4.2: 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.

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. 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.

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 $100 \leq n \leq 1000$: $\hat{\text{CPU}}_{50}$ and $\hat{\text{CPU}}_{90}$. The acknowledgment rate r_a is difficult to estimate in advance and, by construction, $r_a \leq C$. Moreover, the third sub-figure in Figure 4.7 confirms that $r_a \leq C$, allowing us to estimate:

$$\begin{aligned} \hat{\text{CPU}}_{50}(C, n) &\approx 3.48 \times C + 0.0475 \times n + D_1, \\ \hat{\text{CPU}}_{90}(C, n) &\approx 2.14 \times C + 0.0981 \times n + D_2. \end{aligned} \tag{4.8}$$

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 exact coefficients from Equation 4.8 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 4.2, 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 4.8, 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 4.2. As an example, the right-end side of Figure 4.7 shows that *none of the 1000* receivers completed the file download with $C = 25$ Mbps. From Equation 4.5, 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 [ISK25]. 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 4.2, 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 Figure 4.7, 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 4.8 and

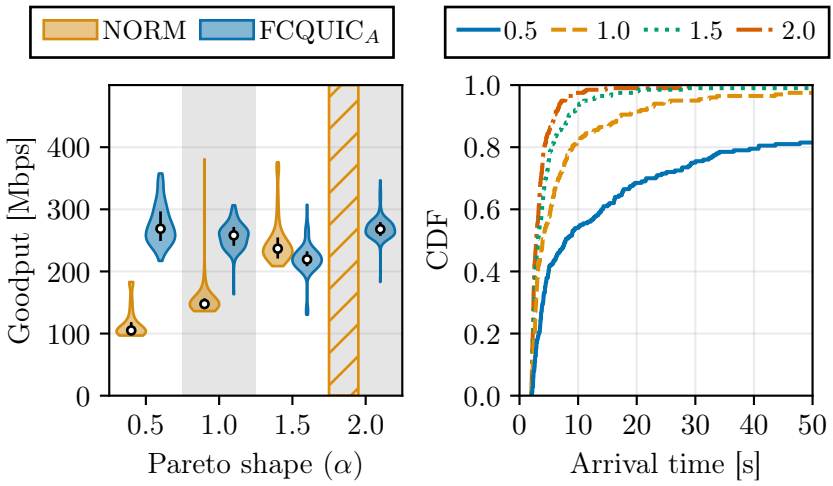


Figure 4.8: Left: FCQUIC$_A$ provides a steady per-receiver goodput independently of the receiver arrival rate. Right: receiver arrival distribution for each α, with $n = 200$ receivers. For $\alpha = 2$, no NORM receiver completed the download.

tailor C (Equation 4.6) 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.

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.

4.4.4 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 Figure 4.8. The right-hand side of the figure shows the distribution of arrival time of the 200 receivers as a function of α. We limit to 200 receivers as it already saturates the 100 Gbps bandwidth of the testbed. 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 during the transfer of a block.

On the contrary, the goodput provided by NORM improves with α: when receivers arrive sporadically (smaller values 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 cache 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. Figure 4.8 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 (3 minutes).

Takeaway. NORM is dependent on the client's arrival rate. Worse, multicast retransmissions for late receivers degrade the goodput of first receivers.

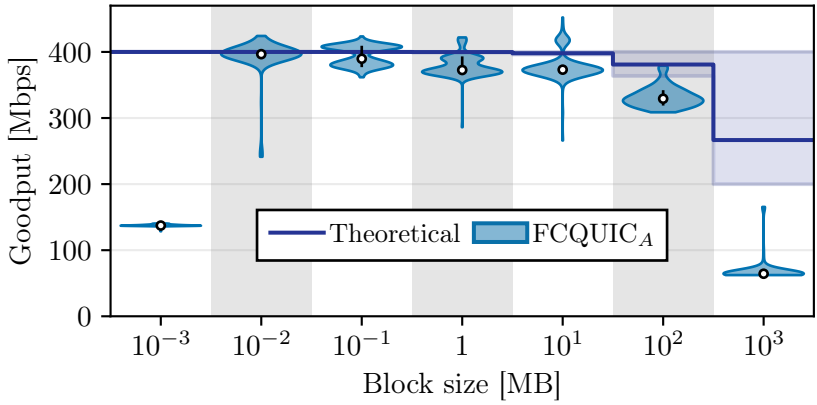


Figure 4.9: Theoretical and measured goodput when varying the block size for $n = 200$ FCQUIC$_A$ receivers. Goodput increases with shorter blocks, whereas too low block sizes incur significant overhead.

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.

4.4.5 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$_A$. Figure 4.9 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

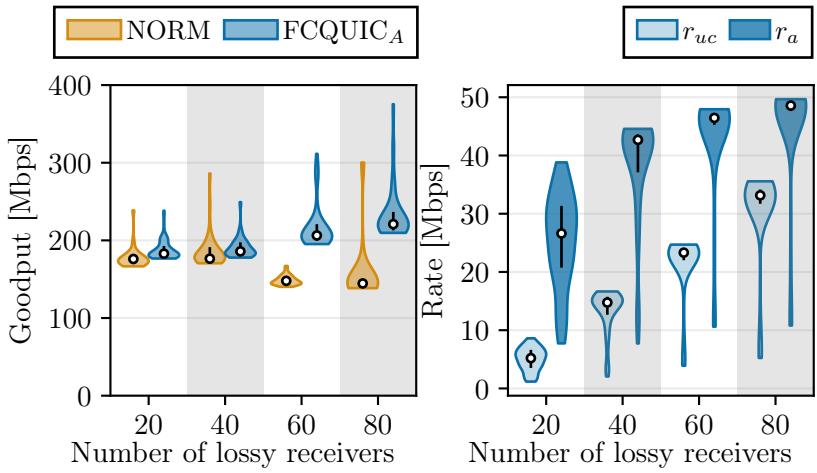


Figure 4.10: Impact of heterogeneous random losses on the goodput of FCQUIC$_A$ and NORM (left), and the resulting unicast (r_{uc}) and acknowledgment rate (r_a) (right).

fits 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 lead to sizes of QUIC packets utilizing the 1500 B MTU, 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 in our testbed.

4.4.6 Impact of heterogeneous losses

Finally, we analyze the impact of losses on the goodput for both FCQUIC$_A$ and NORM in Figure 4.10. We add 0.1 % random and independent losses 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. This approach 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 Gbps link, each multicast retransmission from the NORM source, being replicated to all receivers, increases the utilization of the 100 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 Figure 4.10 shows the distribution of the unicast (r_{uc}, from the source to the receivers) and ack 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 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.

Takeaway. 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.

4.5 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. This chapter builds upon FCQUIC 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. We demonstrate the flexibility of our approach across various scenarios, highlighting 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 fac-

tor 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.

Improvements to this work. This chapter again uses CloudLab as a testbed to evaluate Flexicast QUIC, which limits the scale of our experiments. A first improvement to this work would be to perform the same scalability analysis from Section 4.4.3 across other environments to ensure the results are not overly specific to CloudLab, e.g., testing Flexicast QUIC on real-deployment servers, which are significantly more powerful than those used in this chapter [SH26]. Second, our scalability extrapolation focuses solely on the server's CPU load. In real large-scale deployments involving tens of thousands of receivers, the CPU load may not be the only concern. For such scenarios, the FCQUIC source may not be able to handle numerous open socket descriptors, or the memory footprint of all the opened unicast paths will become non-negligible. Figure 4.4 hinted that, even with 200 receivers, without the acknowledgment strategy from this chapter, UDP socket lookups accounted for a substantial part of the CPU load. Reverting to the single-socket design would remove this concern but would shift the burden of demultiplexing all per-receiver packets to a single task.

Third, our file-rolling system repeatedly sends blocks over *new* QUIC streams, which poses flow control issues in the long run. Indeed, QUIC's specification states that sending/receiving stream data with a given identifier *implicitly* opens all lower-numbered streams of the same type simultaneously. Let us consider the case of a (very) late receiver. With our current mechanism, when the source sends a new block with a significantly large stream identifier, the flow control limits of this new receiver may refuse to open all previous streams, thus resulting in a connection error. An improvement to the current file-rolling mechanism would be to replay streams repeatedly, as allowed by QUIC's specification [RFC9000]. We would expect improved results with this enhanced mechanism, e.g., the *incomplete* blocks in Figure 4.1 would no longer be discarded, allowing the receiver to complete the transfer faster. However, from an implementation point of view, it is much easier to repeat the blocks in successive streams than to repeat the stream itself.

Finally, we use HTTP/3 to initiate clients' requests and to answer with a manifest file describing the file-rolling mechanism (Section 4.3), but the data is sent directly in QUIC streams. The inherent nature of HTTP requires clients to make a GET request to open a new HTTP/3 and QUIC stream so the source can send data for each new block. If a single receiver does not send such a request, the entire communication could be blocked. Another, more elegant solution, would be to use the HTTP PUSH method [RFC9113], which provides cleaner semantics. HTTP PUSH lets the server open additional streams *with-*

out the client's request. On the first opened stream through a GET method, the server first sends PUSH_PROMISE frames with the stream identifiers it will use later on to send this non-requested data. Though HTTP PUSH is part of the HTTP/2 [RFC9113] and HTTP/3 [RFC9114] specifications, it is not deployed in practice, and HTTP/3 stacks, such as *quiche* [QUICHE], do not implement it.

Conclusion. We investigated the *ACK implosion* problem in the practical setting of large-scale file delivery. The design of Flexicast QUIC enables us to reconsider this problem by knowing the receiver set throughout the communication lifetime, a paradigm shift from historical multicast protocol designs. Comparison with the NORM protocol shows that several design choices in FCQUIC positively impact results in our testbed, e.g., unicast retransmissions rather than using the multicast flow, as NORM does. Although we cannot deploy Flexicast QUIC at a larger scale, our CPU analysis suggests that our design and implementation can support significantly more receivers than studied in this chapter.

Analyzing the Optimistic Acknowledgment Attack in QUIC

5

The last chapter showed the benefits of *delaying acknowledgments* from receivers to the source in terms of CPU consumption for a Flexicast QUIC source. The starting point of the chapter was to consider how we could go further and rely solely on *negative acknowledgments* (NACKs) to ensure reliability. Receivers send NACKs only when they detect losses in the packet sequence, thereby reducing the number of messages returned to the source when the loss rate is low. If all receivers report the same loss, the *negative acknowledgment implosion* problem arises, i.e., the source receives the same NACK from multiple receivers. Since QUIC uses a packet number to identify its packets, a gap in the packet number sequence may indicate a loss on the receiving side.

During the initial discussions of a NACK extension to Flexicast QUIC, we identified a specific behavior in the design of QUIC [RFC9000] regarding its packet number sequence. Section 21.4 specifies that:

> An endpoint that acknowledges packets it has not received may cause a congestion controller to permit sending at rates beyond what the network can support. An endpoint **MAY** skip packet numbers when sending packets to detect this behavior. An endpoint can then immediately close the connection with a connection error of type PROTOCOL_VIOLATION.

Due to this behavior, QUIC receivers may incorrectly associate a skipped packet number with a network loss, leading to NACK implosion since all receivers will report the gap. Section 21.4 of RFC 9000 [RFC9000] specifies that this action prevents the *optimistic acknowledgment* attack. In a nutshell, this attack, firstly discussed with TCP [Sav+99], involves sending acknowledgments for packets not yet received to increase the peer's sending rate, ultimately causing network congestion. More importantly, the statement indicates that QUIC endpoints *MAY* skip packet numbers, without requiring it

(*MUST* keyword). Given the large landscape of QUIC implementations, comprising more than 20 code bases [IET25], we investigated which implementations support this security feature, which were exposed, and, in particular, *how* such an attack can be executed against QUIC stacks.

In this chapter, we analyze the optimistic acknowledgment (OACK) attack in QUIC. We evaluate the 16 server implementations from the QUIC Interop Runner [SI20; See25], and find that 11 are vulnerable to the attack. We implement a simple OACK predictor and expose these 11 implementations using in-lab measurements, showing that some stacks send at > 200× the expected rate. We confirm the results by carefully reproducing the OACK attack on a large Content Delivery Network server, and we propose a patch based on existing resilient implementations.

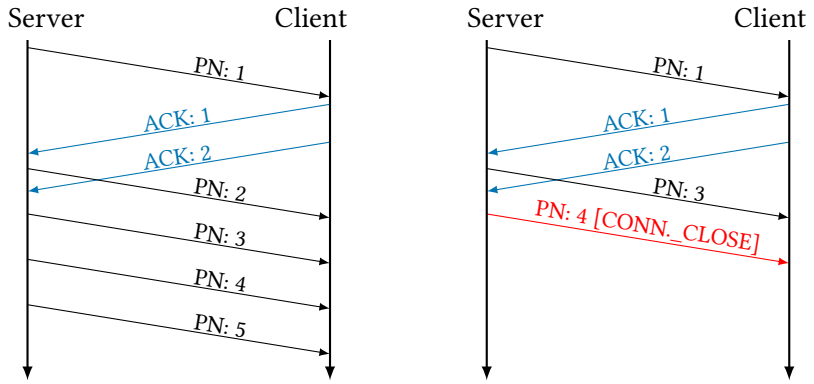
We organize the remainder of this chapter as follows. We first present the optimistic acknowledgment attack in QUIC, and its initial discussion in TCP, in Section 5.1. Section 5.2 explains our simple OACK predictor and explores the vulnerabilities of the QUIC server stacks from the Interop Runner. We validate our results by exploiting the OACK attack against a real CDN server, with the company's consent (Section 5.3). Finally, we detail how to detect and counter the attack in QUIC in Section 5.4.

We published a workshop paper with the results from this chapter in the *Applied Networking Research Workshop 2025 (ANRW'25)*. In addition to the workshop presentation, we presented the results at the QUIC working group meeting at IETF 123. We contacted the maintainers of the 11 vulnerable server implementations. Our discussions led to the correction of 7 of them, including production implementations from Cloudflare (*quiche*), Microsoft (*msquic*), and Alibaba (*xquic*).

5.1 The optimistic acknowledgment attack and countermeasure in QUIC

Reliable transport protocols such as TCP [RFC9293], SCTP [RFC4960], and QUIC [RFC9000] use acknowledgments to confirm the correct reception of data. Each acknowledgment indicates that all data, including the returned sequence/packet number, has been correctly received. Modern implementations of these protocols return acknowledgments at different frequencies. A TCP receiver usually acknowledges every second received packet. The QUIC specification recommends sending an ACK frame after receiving at least two ack-eliciting packets [RFC9000], but the IETF discusses extensions to change this default behavior [ISK25], which was the starting point of the last chapter.

The optimistic acknowledgment (OACK) attack in TCP. These rules apply to honest receivers who seek to receive data reliably. However, in 1999, Savage *et al.* [Sav+99] showed that malicious TCP receivers could ma-



(a) A server not skipping packet numbers. The client guesses that the packet number following 1 is 2, and sends an optimistic acknowledgment for PN=2. Upon reception of this packet, the server increases its congestion window.

(b) A server skipping packet numbers. The client cannot guess in advance that the server will skip packet number 2. Upon reception of an ACK frame with PN=2, the server closes the connection because it detects the attack.

Figure 5.1: Optimistic acknowledgment attack example. *PN* stands for *packet number*. We assume, for simplicity, that the server employs a loss-based congestion control algorithm, such as CUBIC.

nipulate their acknowledgments to launch denial-of-service attacks against servers. In a nutshell, to force the server to send data as quickly as possible, a malicious receiver could send optimistic acknowledgments that ignore packet losses (i.e., never return duplicate acknowledgments of SACK blocks [RFC2018]) or worse, send acknowledgments for data that it did not yet receive. Savage *et al.* proposed extensions to TCP to solve this problem, but they have not been adopted. In 2005, Sherwood *et al.* demonstrated that the attack was possible against different TCP implementations [SBB05]. Several solutions were suggested to prevent such attacks against TCP using cumulative *nonces* [Sav+99] and by randomly skipping sequence numbers [SBB05]. In 2016, Schaub and Trey released a user-space implementation of TCP that performs this attack [ST16]. Laraba *et al.* proposed a solution to protect TCP servers by tracking all flows and identifying misbehaviors [Lar+21].

The optimistic acknowledgment (OACK) attack in QUIC. This attack was known during the QUIC protocol standardization, and countermeasures were discussed. Since QUIC encrypts most of the packet header and data, a firewall cannot analyze the incoming QUIC packets and block optimistic acknowledgments, in contrast to TCP. For simplicity, let us consider a client-server connection in which the client requests a large file. The client is malicious and sends optimistic acknowledgments to the server to increase its transmission rate. We assume that the client is not interested in the trans-

Implementation	quic-go v0.50.0	ngtcp2 v1.11.0	mvfst v2025.03.03	quiche v0.23.4
(1) Skip PN	✓	✗	✗	✗
(2) Correctness	✓	✓	✓	✗

Implementation	kwik v0.10.1	picoquic 6304c2e9cc35	aioquic v1.2.0	neqo v0.12.2
(1) Skip PN	✗	✓	✗	✗
(2) Correctness	✗	✓	✗	✓

Implementation	nginx 145b228530c3	msquic v2.3.9	xquic v1.8.2	lsquic v4.2.0
(1) Skip PN	✗	✗	✗	✓
(2) Correctness	✓	✓	✓	✓

Implementation	haproxy v3.1	quinn v0.5.9	s2n-quic v1.52.0	go-x-net d18fa4cfbd84
(1) Skip PN	✗	✓	✓	✗
(2) Correctness	✓	✓	✓	✓

Table 5.1: 11 of the 16 server implementations available from the QUIC Interop Runner are vulnerable to optimistic acknowledgment attacks. We report the version used for the measurements (vX.Y.Z) or the *docker* image hash. Implementations are vulnerable if (1) they don't skip packet numbers. Worse, (2) implementations that do not assess the correctness of the ACK frames expose themselves to straightforward OACK attacks. The maintainers of the 11 implementations confirmed their vulnerability to the OACK attack. Patched implementations, as of the time of writing, are underlined.

mitted data; its sole objective in requesting the file is to attack the server.

Figure 5.1a illustrates this attack on QUIC. The malicious client sends acknowledgments for packets that it has not yet received. The server responds to these acknowledgments by increasing its congestion window and transmission rate, since all data in flight has been correctly received. By sending acknowledgments in advance, the malicious client *hides potential losses* from the server, thereby artificially increasing the server's congestion window without accounting for the underlying network conditions. Such an attack can impact both the server and the network. A coordinated attack targeting the same server could saturate its network and affect legitimate clients. In a nutshell, a QUIC server is vulnerable to the optimistic acknowledgment (OACK) attack if the client can infer the server's sequence of packet numbers. By knowing this sequence, the client can predict packets it has not yet received and hide the network losses, thus increasing the server's bit rate.

Skipping packet numbers. The client must infer the server's packet number sequence to optimistically acknowledge packets sent by the server. This must be done carefully to avoid sending an ACK frame for unsent pack-

ets. Indeed, the majority of tested implementations close the connection when they receive an acknowledgment for an unsent packet (Line *(2)* in Table 5.1). For example, the client could simulate the CUBIC congestion window update mechanism to anticipate the server's packet transmissions. QUIC [RFC9000] includes a mechanism to prevent malicious clients from attacking the server with OACK: the possibility to *skip* packet numbers. Section 21.4 of the QUIC specification [RFC9000] mentions that *an endpoint MAY skip packet numbers when sending packets to detect this behavior* (i.e., OACK). *An endpoint can immediately close the connection with a PROTOCOL_VIOLATION connection error.* There was no consensus on the mandatory skipping of packet numbers with the IETF QUIC working group to prevent these attacks, and Section 21.4 of RFC 9000 includes this as a possible feature. If the server skips some packet numbers when emitting packets, the client cannot guess the correct sequence. Figure 5.1b illustrates the server skipping packet number 2, which the client cannot guess before receiving packet number 3. Since the server never used packet number 2 for data transmission, there is no valid reason to receive an acknowledgment for it. As a reminder, packet numbers in QUIC are *unique* and must be monotonically increasing. As such, when the server skips packet number 2 and sends a packet with number 3, it will never reuse lower-numbered packet numbers for the duration of the connection. The server can safely close the connection upon receiving an ACK frame for packet number 2, thereby preventing the OACK attack. By *randomly* skipping some packet numbers, the server can quickly identify malicious receivers and potentially close the connection.

11 of 16 server implementations are vulnerable to the OACK attack. We use the QUIC Interop Runner [SI20; See25], an open-source software that automatically runs interoperability tests among participating QUIC implementations; any implementation can join the project. All implementations expose a Dockerfile with a common interface for testing. We run each server implementation in the runner to collect the packets it emits during a 30 MB file transfer. We observe that the server skips packet numbers when gaps appear in the sequence of packets it emits during the transfer. Out of the 16 tested servers, 11 do *not* skip packet numbers, including the implementations from Meta (*mvfst*), Microsoft (*msquic*), Alibaba (*xquic*), and Mozilla (*neqo*). The maintainers of these implementations confirmed that they lack support for packet number skipping. We noted that *mvfst* and *ngtcp2* start at a random packet number in the 1-RTT epoch and subsequently use continuous packet numbers.

Assessing acknowledgments. During this measurement, we noticed that several implementations do not assess the correctness of received acknowledgments, i.e., the server does not verify if the received ACK frames contain ranges of packets that were not (yet) sent by the server. For example,

the client acknowledges packets 0..20 while only 12..16 are in flight. Such a server would extract the 12..16 range without accounting for the fact that the 17..20 range corresponds to packets that were not sent. By skipping this verification step, the server is exposed to straightforward OACK attacks, as the malicious client does not need to accurately predict the packet number sequence.

Quiche, kwik, and aioquic do not verify the correctness of received acknowledgments. To identify which implementations are vulnerable to this condition (2) in Table 5.1, we create a simple client continuously acknowledging the $0..10^6$ range. The 10^6 value is chosen to align with the initial flow control limits of most participating implementations. We observe that *quiche*, *kwik*, and *aioquic* do not assess the correctness of received ACK frames. We reported this issue to the maintainers of these three implementations. While these implementations are vulnerable to straightforward OACK attacks (e.g., by acknowledging large ranges of packets, such as $0..10^6$), we include in the study implementations that verify the correctness of received acknowledgments. To perform the attack on such implementations, the malicious client must schedule optimistic acknowledgments in time to send them to the server. For this purpose, we design an OACK predictor, which determines the correct time to send such ACKs.

5.2 Implementations' vulnerabilities

In Section 5.2.1, we design our OACK predictor and implement it in a client application based on Cloudflare *quiche*. We selected *quiche* solely due to our preference for the implementation. Then, we conduct the OACK attack against the 11 vulnerable server implementations in the controlled environment described in Section 5.2.2.

5.2.1 Optimistic acknowledgment predictor

The optimistic acknowledgment predictor aims to predict the packets the server has sent before they are actually received. Our OACK predictor must fulfill two properties: (i) Generate acknowledgment ranges faster than if the corresponding packets were actually received; (ii) The server must not receive ranges for unsent packets. Designing this predictor in QUIC is more challenging than in TCP. Indeed, Section 3 of RFC 9293 [RFC9293] states that upon receiving an invalid acknowledgment (such as an OACK), a TCP server should ignore the ACK and send an empty packet with the correct sequence number without sending a RST to the client. This design adheres to Postel's principle: *Be conservative in what you send, and liberal in what you accept.* OACK predictors in TCP could rely on this mechanism to synchronize the

OACK scheduler with the server when acknowledgments are sent too quickly. Indeed, in [SBB05], the authors use a simple algorithm that can overestimate the rate at which the server sends data. When this occurs, and the server sends an empty packet with the correct sequence number it expects to receive, the malicious client restarts its prediction from scratch without breaking the connection. This conservative approach in TCP is mandatory since sequence numbers can be reused during the lifetime of the connection. Otherwise, a delayed, valid acknowledgment could be misinterpreted by the server as an optimistic acknowledgment and as an attempt to produce the attack. Again, such an interpretation is impossible in QUIC since packet numbers are unique during the connection lifetime.

As such, a QUIC server immediately closes the connection if it detects invalid acknowledgments, necessitating more sophisticated predictors. Al-

Algorithm 3: Simple QUIC OACK predictor.

i_d: aggressiveness of increase of d
l: number of packets before starting OACK
d: OACK increase. Init: 0
n_m: maximum received packet number
n_0: first packet number with data. Init: None
n: received packet number
Result: Potential OACK ranges to send to the peer

```
1 if n_0 is None then
2 |   n_0 ← n
3 if n_0 + l > n or n < n_m then
4 |   return No range;
5 n_m ← n;
6 d ← d + i_d;
7 return n_0..n + d;
```

gorithm 3 presents our optimistic acknowledgment predictor for QUIC, introducing two parameters. To prevent the client from injecting OACKs too early in the connection, we delay the start of the attack by l received packets (line 3). For simplicity, we denote n_0 as the first packet number used by the server to deliver data. Typically, $n_0 = 0$, but we observed that *mvfst* and *ngtcp2* use a random $n_0 > 0$ in the 1-RTT packet number space. The core idea of the predictor is to acknowledge each received packet with an increasing delta. Whenever the client receives a new packet with a packet number n, it sends an ACK frame acknowledging packets $n_0..n + d$. As such, the client acknowledges d more packets that have not yet been received. The trick is to continuously increase d by i_d because the attack induces congestion in the

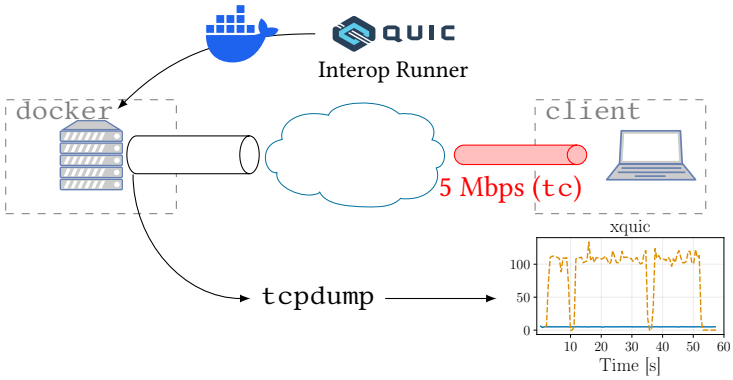


Figure 5.2: Virtual testbed running on a single machine. A Linux bridge connects the two namespaces.

network, while the client continues to increase the server's bit rate.

The spirit of this constant growth is to simulate the *additive increase* of loss-based congestion control algorithms [RFC9438]. Let us assume that our client receives packets at a rate of 5 Mbps in steady state. If $i_d = 0$, we constantly acknowledge d more packets than what we actually receive. In that case, the predictor is simply acknowledging packets faster than it should, given its limited 5 Mbps bandwidth. To simulate an infinite amount of available bandwidth for the attack, i_d ensures that we continually acknowledge more packets, in the *additive increase* spirit of these congestion control algorithms.

Flow control. Flow control limits ensure that the sender does not send more data than what its peer can handle. These limits are updated by the receiver during the lifetime of the connection. To avoid blocking the server's transmission due to flow-control limits, our malicious client sets its flow-control limits to arbitrarily large values at the beginning of the connection through the MAX_DATA and MAX_STREAM_DATA frames.

5.2.2 Exploring QUIC stacks' vulnerabilities

In this section, we carry out the OACK attack on the 11 vulnerable QUIC server stacks from Table 5.1. During the following experiments, we set $i_d = 4$ and $l = 400$. The objective of this work is to assess whether we can produce the optimistic acknowledgment attack on theoretically vulnerable stacks, not to optimize the attack. As such, we do not explore other sets of values.

Virtual testbed. We use the virtual testbed illustrated in Figure 5.2 on a single machine to encourage the reproducibility of the experiments. We work with NPF [Bar24] to orchestrate and reproduce our experiments. We leverage

the *docker* images from the QUIC Interop Runner [SI20; See25] to run the unmodified servers in their namespace (`docker`). Our malicious client runs in its namespace (`client`), and a Linux bridge connects these two namespaces. Using `tc`, we add a 5 ms delay in both directions on the main namespace. We add a 5 Mbps bandwidth limit towards the client to show the attack's impact in a constrained environment. The client requests a 600 MB file via an HTTP/3 request, which is sufficient to demonstrate the optimistic acknowledgment attack. We limit each experiment to 60 s. Our malicious client uses small initial flow-control limits of 10 MB and increases them based on the optimistic acknowledgments it sends. We measure the server's bit rate at its egress, i.e., by running `tcpdump` within the `docker` namespace.

Our OACK predictor tricks the 11 servers into increasing their transmission bit rate up to 300× compared to normal behavior. Figure 5.3 shows the server bit rate with a benign client and with OACK enabled on the client. The duration and success of the attack vary across QUIC implementations. For example, *msquic* closes the connection because the OACK predictor sent an acknowledgment for an unsent packet after ~2 s of the attack. However, over this period, the server sends up to 1.5 Gbps of traffic, a 300× increase in the actual network bandwidth of 5 Mbps. Other implementations, such as *xquic* and *mvfst*, send between 100 and 300 Mbps traffic to the network during 50 s and 5 s respectively. In conclusion, Figure 5.3 assesses that our OACK predictor can attack these servers by making them send data too quickly into the network despite its simplicity.

The client transmission pattern does not indicate any malicious behavior. Figure 5.4 analyzes the client transmission bit rate (right sub-Figure) when sending OACKs. Our predictor only sends OACKs (with extended ranges) when it receives a packet from the server. As such, the bit rate towards the server follows normal behavior. Because QUIC packets are end-to-end encrypted, a middlebox cannot detect the attack by analyzing the content of the acknowledgment frames the client sends. We even note that without OACK, the client's transmission bit rate is higher: the client sends larger ACK frames with more ranges due to the gaps induced by the congestion losses.

CUBIC vs BBR on the server. Our OACK predictor is based on the hypothesis that servers use loss-based congestion control algorithms (CCAs), such as CUBIC [RFC9438]. Such algorithms never decrease their congestion window unless they detect losses based on (the lack of) acknowledgments from their peer. However, non-loss-based CCAs, such as BBR [CSB25], employ a different mechanism to adapt the congestion window. In 2019, BBR accounted for ~40 % of Internet traffic [Mis+19]. A BBR server may proactively decrease its congestion window during a *probing phase*. If a client cannot determine when the server enters the *probing phase*, it may send an acknowl-

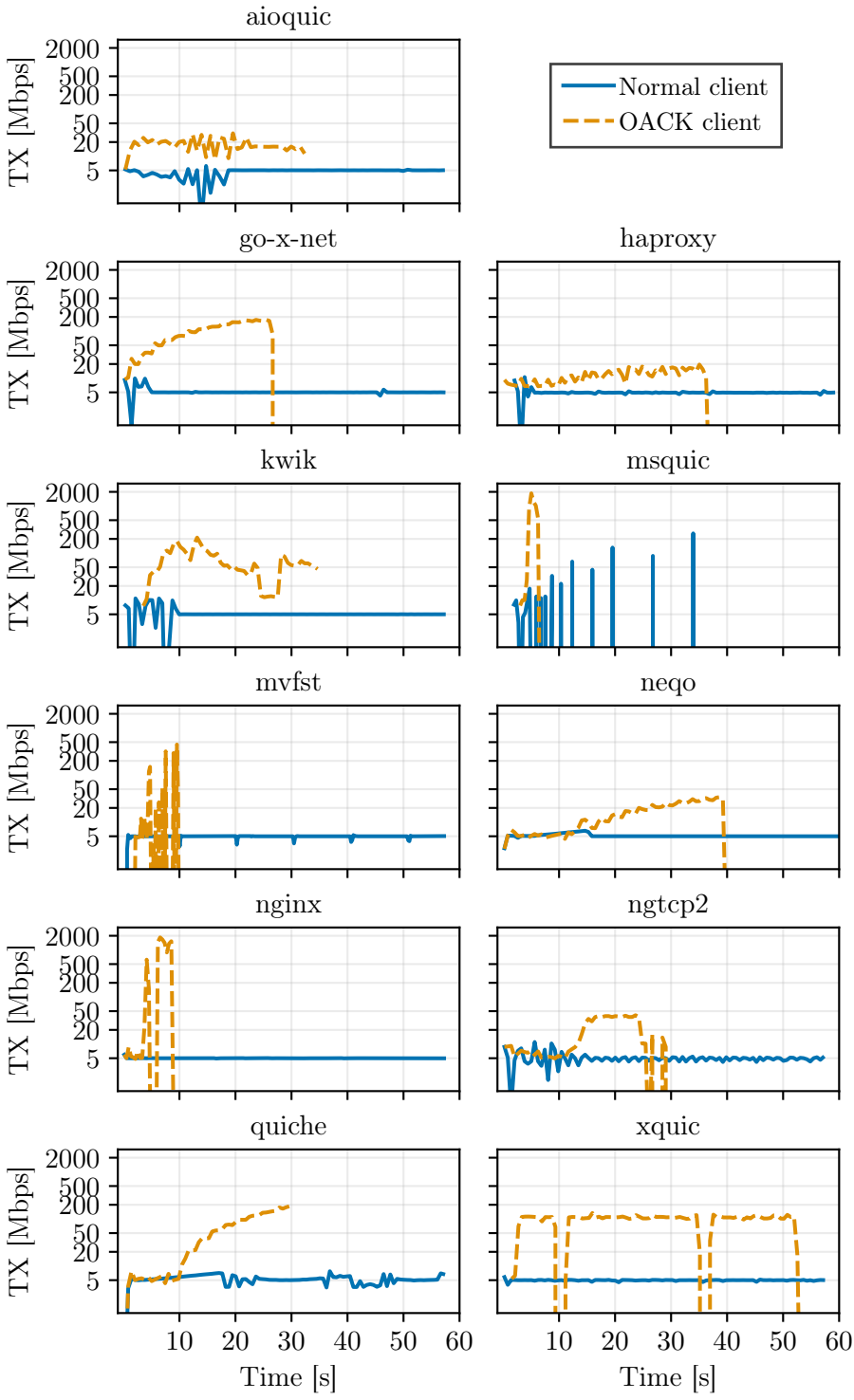


Figure 5.3: Transmission bit rate measured on the server egress with a client limited at 5 Mbps reception bandwidth, using the vulnerable servers' implementations taking part in the QUIC Interop Runner.

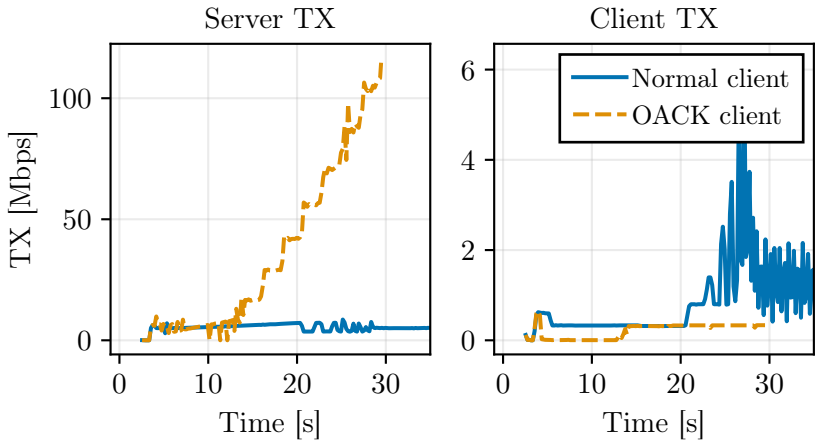


Figure 5.4: Server (left) and client (right) bandwidth of our OACK client against a *quiche* server.

edgment for an unsent packet due to reduced throughput. Figure 5.5 shows the impact of using BBRv2 instead of CUBIC with an OACK client and the *ngtcp2* server. The server's congestion window temporarily decreased without seeing losses, but our simple OACK predictor did not account for this behavior. As such, it continued to increase the range of acknowledged packets. Upon reception of an ACK for an unsent packet, the *ngtcp2* server closed the connection. While using BBR initially appears to be a good remedial strategy, a more sophisticated predictor could identify such *probing phases* to attack a BBR server. We do not attempt to develop such a more sophisticated predictor, as we believe it is outside the scope of this chapter.

Figure 5.6 explores the same scenario with *quiche*, which does not assess the correctness of the received ACK frames (condition (2)). As mentioned, the server does not close the connection when our predictor sends acknowledgments for unsent packets. Interestingly, the server's bit rate with BBRv2 is more than twice that with CUBIC. Because BBR estimates the available bandwidth from the rate at which its in-flight data is acknowledged, the optimistic acknowledgments inflate this delivery-rate estimate, driving the server to raise its pacing rate. This result again highlights the risks of not verifying the correctness of received ACK ranges.

5.3 Attacking a deployed Content Delivery Network server

In the previous section, we focused on generating the attack in a controlled environment. Although we treated the server as an opaque box, production QUIC servers may use configurations and features that differ from those in the

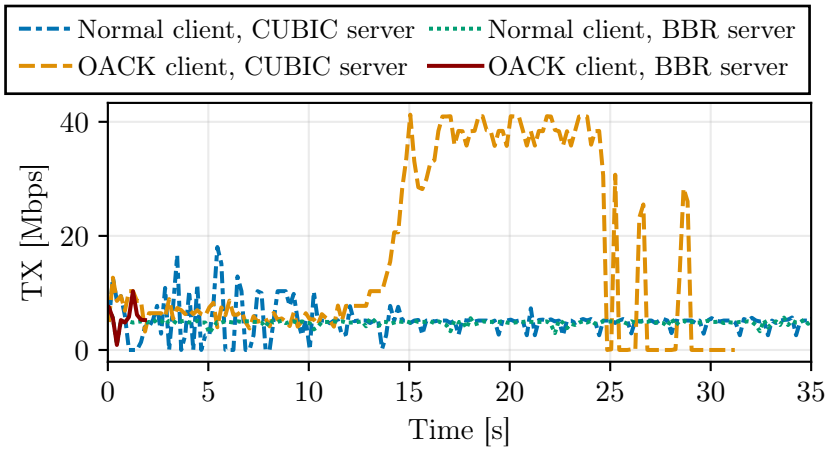


Figure 5.5: Bandwidth measured on the *ngtcp2* server when using CUBIC against BBRv2.

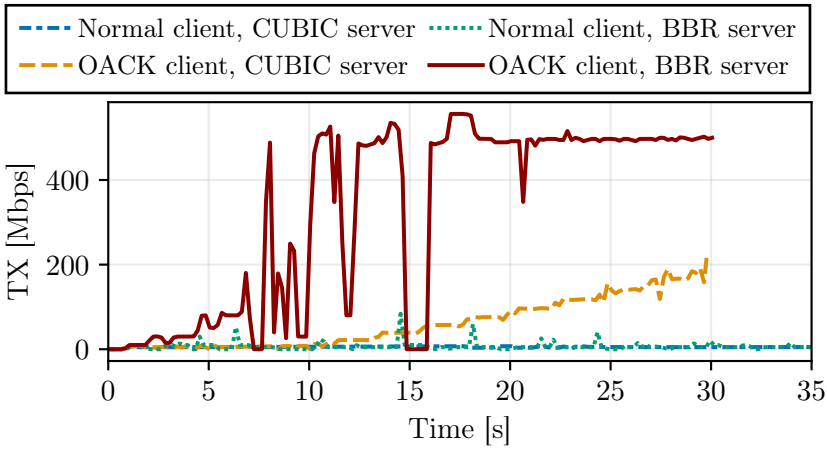


Figure 5.6: Bandwidth measured on the *quiche* server when using CUBIC against BBRv2.

QUIC Interop Runner Docker images. For example, while Cloudflare *quiche* implements the Connection Migration feature of QUIC [RFC9000], Cloudflare servers consistently disable it [BP25]. We now assess whether we can produce the optimistic acknowledgment attack on a deployed production QUIC server. Because we are targeting deployed infrastructures serving customers, we must be careful regarding the scale of the attack. We performed the attack only after obtaining explicit consent from the company that manages the deployed service. As a result, we only performed the optimistic acknowledg-

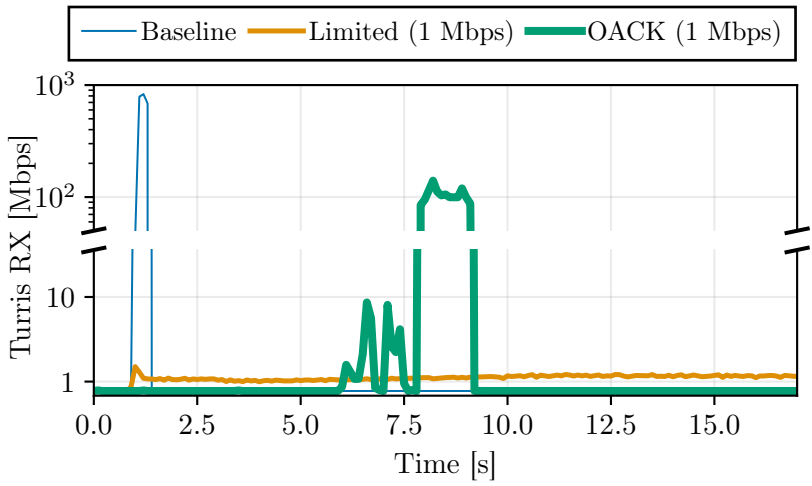


Figure 5.7: Bandwidth measured on the Turris access point. The client makes an HTTP/3 request to our website hosted on a large Content Delivery Network.

ment attack on a single target that agreed: a large Content Delivery Network (CDN) using one of the vulnerable QUIC stacks from Table 5.1.

We host a static website on the CDN, which consists of high-quality images. This service is provided by the CDN and handles the website's network stack, security, and server scaling. As a result, the CDN protects our website against standard denial-of-service attacks. We did not modify any network or security-related configuration. The website is accessible via HTTP/2 (above the TCP/IP protocol suite) and HTTP/3 with QUIC, although we focus only on HTTP/3 and QUIC in this chapter. The experiment involves downloading eight images, totaling 17 MB.

The malicious client is connected to the Internet via a Turris Omnia access point to assess the impact of the OACK attack while remaining cautious. Indeed, we limit the bandwidth from the Turris to the client to 1 Mbps using `tc`. Because we cannot directly measure the server's transmission bit rate since we do not control its network stack, we measure the bandwidth at the ingress of the Turris Omnia, before application of the `tc` limit. We use a hard 1 Mbps limit to better reflect the attack's impact without overloading the CDN network. Measurements without the `tc` limit in this setup achieved a maximum throughput of ~700 Mbps. As such, we argue that performing the optimistic acknowledgment attack to force the server into sending less than 700 Mbps of traffic should not harm the Content Delivery Network servers. The CDN explicitly validated our experimental setup prior to any measurement.

The production QUIC stack from the Content Delivery Network is

vulnerable to the optimistic acknowledgment attack. We use our predictor from Algorithm 3 with $l = 400$ and $i_d = 10$. Figure 5.7 reports the results. The *Baseline* curve shows the expected throughput *without* the 1 Mbps bandwidth limit, indicating that we can expect to receive data up to 700 Mbps. With the bandwidth limit and a benign client (the *Limited* curve), the reception bit rate does not exceed 1.1 Mbps, showing that the server correctly adjusts its transmission bit rate to the bottleneck link, which is between the Turris and the client. However, the *OACK* curve indicates that our malicious client correctly increases the server's transmission bit rate due to OACKs; we receive up to 100 Mbps of traffic despite the hard 1 Mbps theoretical limit. To avoid taking the risk of *actually* attacking the CDN, we do not attempt to reach the 700 Mbps baseline bit rate. However, we argue that the results from Figure 5.7 show the potential of the attack if executed more aggressively against a deployed server.

The next section discusses how to counter the optimistic acknowledgment attack and the patch we suggest based on already robust implementations. Let us note that, once again with their agreement, we performed the attack on the CDN after applying the patch that implements packet number skipping; the CDN server was no longer vulnerable to the attack.

5.4 Preventing the optimistic acknowledgment attack

As indicated in Table 5.1, five implementations, at the time of measurement, were robust to the optimistic acknowledgment attack. These stacks implement the countermeasure described in Section 21.4 of RFC 9000 [RFC9000]. To protect a QUIC host against OACKs, the implementation must (i) assess the validity of the received acknowledgment and (ii) randomly skip packet numbers during the transfer.

Assessing acknowledgments' validity. To assess the validity of received ACK frames, QUIC endpoints must match the acknowledged ranges (of packet numbers) to the packet numbers sent. For example, Cloudflare *quiche*[1] iterates over all unacknowledged packets, checking whether their packet numbers fall within the received ranges, without assessing their validity. Assuming the endpoint uses continuously increasing packet numbers (the case of skipped packet numbers is analyzed later), a correction consists of verifying whether the ranges contain packet numbers higher than the maximum packet number the endpoint sent.

Skipping packet numbers. As stated by RFC 9000 [RFC9000], QUIC endpoints *may* skip packet numbers to prevent the optimistic acknowledgment attack. It is sufficient to *randomly* skip a few packet numbers during the

[1] Version 0.23.5, released on April 2, 2025

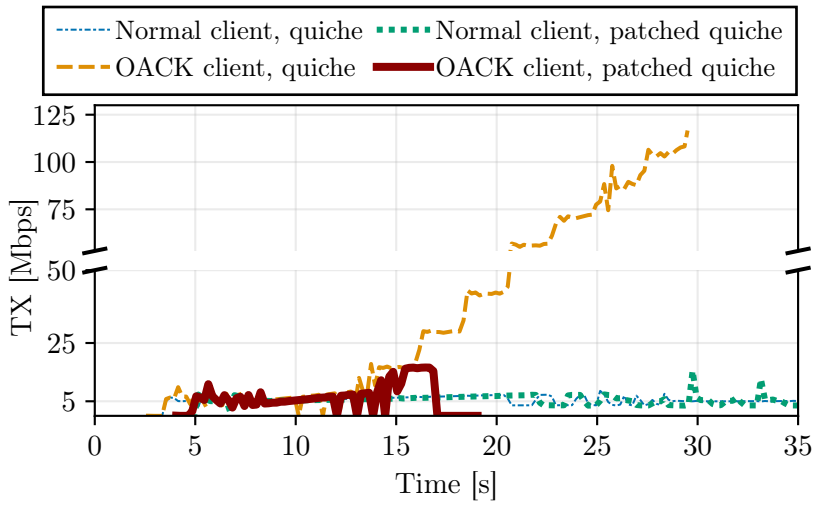


Figure 5.8: Throughput on the fixed *quiche* server.

connection to limit the overhead induced by the increased number of ranges caused by the gaps. For example, *quic-go* randomly skips a packet number at least every $128 \times 1024 = 131,072$ packets; *picoquic* randomly skips a packet number uniformly in the range $[3, 256^{1+n_a}]$ where n_a is the number of already skipped packet numbers. To detect an OACK attack, both implementations generate dummy, unsent packets associated with the skipped numbers. When the endpoint receives an ACK from its peer, it checks whether the ranges acknowledge the dummy packet; if so, they close the connection with a protocol error.

Patching *quiche*. We patched Cloudflare *quiche* to address ACK correctness and packet-number skipping. It adds 187 and deletes 16 lines of code. For simplicity, we use the same mechanism as implemented in *quic-go*. For each outgoing packet, we skip a packet number with a uniform probability of $\frac{1}{131,072-n_{ls}}$, where n_{ls} is the number of packets we sent since the last packet skip. This value is reset to 0 whenever we skip a packet number; otherwise, it is increased by 1. The probability of skipping a packet number increases with time as long as we do not skip one.

We run our OACK client against this modified *quiche* server in the same setup as in Section 5.2.2. Figure 5.8 highlights that the patched server is not vulnerable anymore to the OACK attack. When the malicious client enters the OACK phase, the skipped packet number triggers the server to close the connection after 17 s. The corrected server transmits at most 18 Mbps over its network before detecting the attack.

Frequency of packet number skipping. The server could detect the

attack more quickly by skipping packet numbers more frequently, at the cost of increased overhead and a larger number of ranges acknowledged in ACK frames. Indeed, ACK frames contain ranges of received packets. When the client detects a gap in the packet number sequence, it must include an additional range in its acknowledgment frame. This increases the protocol's overhead because ACK frames contain more bytes. However, because *ACK ranges* use relative numbers and variable-length integer encoding [RFC9000], the overhead of adding such new ranges in the case of skipped packet number is of a few bytes only.

Skip more than one packet number. During our discussions with the maintainers of *xquic* (see Section 5.5), the following question was raised: *Should we skip more than one packet number at a time?* Although we did not evaluate its impact, we do not believe this mechanism would add robustness against the OACK attack. Skipping a single packet number is efficient enough to detect the attack. The probability of a malicious client detecting a randomly skipped packet number is already extremely low. Skipping more than one number would only slightly further lower this probability. We argue that it would be more beneficial to increase the frequency of skipping a single packet number than to skip numerous numbers at once.

Impact on the CPU. Skipping packet numbers can incur additional CPU usage, as the emitter may store state for skipped packet numbers to detect the attack. In our patch to *quiche*, the emitter stores dummy entries associated with these skipped packet numbers. When processing ACK frames, it verifies whether the received ranges contain a dummy entry, and releases it once subsequent packet numbers have been acknowledged. This additional processing occurs every time the emitter skips a packet number, which is rare (our patch skips one packet number every 131,072/2 packets on average). As a result, we do not expect a significant increase in CPU load with the packet number skip feature.

5.5 Discussion

This chapter analyzes the exposure of the QUIC protocol to the optimistic acknowledgment (OACK) attack, originally discussed for TCP [Sav+99]. While QUIC's specification [RFC9000] suggests skipping packet numbers to prevent the attack, we showed that 11 of the 16 server stacks participating in the QUIC Interop Runner [SI20; See25] are exposed to OACKs. Using our simple OACK predictor, we demonstrated the attack's feasibility on these implementations and its efficiency in both an emulated and a real network. Because QUIC packets are end-to-end encrypted, middleboxes and firewalls cannot identify an OACK attack upstream. Worse, a third party cannot detect the attack because the client's acknowledgment rate appears benign. We proposed a ~180-

line patch for *quiche*, one of the most vulnerable stacks, and demonstrated its resilience to the OACK attack when the recommendations are correctly implemented. QUIC servers become robust to the optimistic acknowledgment attack by skipping packet numbers. While QUIC's specification [RFC9000] exposes this feature as a possibility, we argue that *MAY is not enough* and *QUIC servers SHOULD skip packet numbers*.

Impact of this work on the QUIC ecosystem. Following this work, we discussed with the maintainers of all vulnerable stacks; our presentation at IETF 123 further raised awareness and increased interaction with them. As illustrated in Table 5.1, seven of the eleven vulnerable stacks implemented the packet number skip feature to become robust to the optimistic acknowledgment attack following our work. Cloudflare *quiche* also implemented the correctness verification of received acknowledgments. Although the optimistic acknowledgment attack has been known for several years, we argue that this work has raised awareness of the attack's feasibility and that the countermeasure remains effective despite its simplicity. At the time of writing, multiple maintainers have decided not to implement packet number skipping in their implementations, including *neqo* (Mozilla). During our discussions, their core maintainers stated that the primary use of the implementation is for client web browsers. As such, they stated that the probability that a server will perform an attack against a *neqo*-based client is extremely low.

The optimistic acknowledgment attack in TCP. This chapter focused on the OACK attack in QUIC, but the attack exists, and was initially discussed, in TCP [Sav+99]. While QUIC provides a native mechanism to prevent such an attack, it is much trickier in TCP because TCP's sequence number is directly correlated to the offset of the byte stream. Skipping sequence numbers is interpreted as packet loss or gaps in the byte stream; this is similar to implementing the prevention mechanism in QUIC by skipping stream offsets. At the time of writing, discussions to develop a prevention mechanism for the OACK attack in TCP without fundamentally changing the protocol are ongoing. However, interest in designing a solution is low. Indeed, prominent Internet actors do not consider this attack a threat to their infrastructure (e.g., Mozilla is client-oriented). Indeed, the CDN that agreed to allow us to perform the OACK attack against its QUIC servers did not record in its security log any attempt to perform the optimistic acknowledgment attack in TCP. Finally, implementing a countermeasure in TCP typically requires modifying the kernel, which is trickier and more complex to deploy than modifying a QUIC user-space stack.

Improvements to this work. First, we could strengthen this chapter by conducting additional measurements on deployed servers. This would require the explicit approval of the companies that manage these servers, which would have slowed the development of this chapter. We argue that demon-

strating the feasibility of the optimistic acknowledgment attack across eleven vulnerable implementations is the primary outcome of this chapter, as we believe that the companies behind most of these implementations use the same libraries on their production servers. Second, the OACK predictor is simple and performs poorly against servers using BBR as their congestion control algorithm. Again, we believe that the principal outcome of this chapter was to motivate maintainers of QUIC stacks to implement the packet number skipping [RFC9000].

Ethical considerations. We contacted the lead developers of the vulnerable implementations before submitting the paper linked to this chapter. All of them either implemented packet-number skipping or explicitly stated that they do not want to do so. Our patch for *quiche* requires only 187 lines, including comments and updates to existing tests, indicating that the patch is easy to implement. We conducted most of our experiments in our controlled testbed. The real network measurements were performed carefully and with the target CDN's agreement. We constrained our network during this experiment and ensured it did not exceed our *real* bandwidth share during the attack. Finally, the optimistic acknowledgment attack is well known and is even mentioned in the QUIC specification [RFC9000].

Discussion and Perspectives

6

We concluded each of the preceding chapters with suggestions for improvements. In this chapter, we go a step further by discussing how the contributions of this thesis pave the way for future research and industrial contributions. We also step back from the presented work to assess its fit within today's research landscape.

6.1 Network-layer FEC protection

The system introduced in Chapter 2, HIRT, suggests transparently protecting traffic using Forward Erasure Correction, thereby improving the latency of applications in the presence of packet losses. After benchmarks demonstrating its high-speed packet-processing capabilities, we assessed our design in a real network using Starlink, which exposes non-congestion-induced losses [Mic+22b]. HIRT's deployment is conditional on the support of IPv6 Segment Routing, which is often confined to intra-domain networks for controlled traffic engineering purposes, though SRv6 is being increasingly deployed in such networks [ID25].

Recovering congestion-induced losses. Until now, we considered using HIRT to recover from non-congestion-induced losses, such as those exposed by Starlink [Mic+22b]. However, most losses observed on the Internet are attributable to congestion events. In a nutshell, we can consider these events to occur when the input traffic on a node exceeds what the node can process or push onto the outgoing link. However, these congestion events are often very short, especially in data-center networks, and are usually referred to as *incast* [Che+09].

Responding to congestion by generating FEC repair packets is counterproductive, as it further increases bandwidth consumption and thus the load on routers, and is often a major criticism of FEC on the Internet. However, bandwidth usage decreases at the end of an incast, often well below the maximum capacity, leading to performance collapse in data centers [Che+09]. One could leverage HIRT to generate repair packets *after* the incast. As such, these repair symbols can be used to recover the potential losses due to the congestion, without the need for costly retransmissions. This method requires

nodes to detect the start and end of incast events, which is non-trivial. Moreover, recovering congestion-induced losses requires particular attention to the congestion-control state of end-hosts, as hiding these losses from these algorithms is not recommended [RFC9265]. This method could still be combined with Explicit Congestion Notification (ECN) marking on recovered packets to propagate the congestion event to these hosts. Though we concede that this extension is not trivial to design and implement, we believe that it is worth investigating. As discussed in section 2.11, this could be further combined with L4S-capable networks [RFC9330].

6.2 Flexicast QUIC

Flexicast QUIC (FCQUIC) extends QUIC [RFC9000] and Multipath [Liu+26] to provide multicast support at the transport layer. We combine multicast and unicast within the same connection, allowing hosts to seamlessly transition between the two communication methods depending on the quality of their connection. In Chapter 3, we presented the design of Flexicast QUIC and showed that our server software implementation could sustain 80 Gbps of cumulative multicast traffic with 1000 receivers. We also demonstrated the flexibility of FCQUIC by dynamically breaking and degrading multicast connectivity for a subset of receivers, which fell back to unicast until they recovered sufficient multicast access, without impacting the quality of the application (a video stream). In Chapter 4, we extended this work by investigating the ACK implosion problem, inherent to reliable multicast. We showed, using the same testbed, that the source can adapt the acknowledgment rate of receivers by dynamically advertising an acknowledgment delay. Furthermore, we showed that this acknowledgment rate is the primary factor influencing the CPU load of the FCQUIC source, allowing us to extrapolate beyond the limitations of our testbed. We again discussed aspects to improve both chapters in their conclusion. That being said, these chapters also raise numerous research questions that we did not address in this thesis and reopen other historical concerns about multicast that must be answered to consider large-scale deployment of Flexicast QUIC.

The initial design of multicast, almost 40 years ago, assumed that the source should not maintain state proportional to the number of receivers, i.e., it does not know *how many* receivers there are or *who* they are. The failure of multicast to achieve wide deployment stems from multiple factors, including routing complexity, inter-domain policy issues, and the lack of a reliable transport protocol. Meanwhile, the rise of Content Delivery Networks (CDNs) showed that, even if relying solely on unicast protocols is not optimal in terms of traffic distribution, *it works*. As such, maintaining per-receiver state on the FCQUIC source should not be considered harmful, as this is al-

ready the norm in today's Internet. While Flexicast QUIC does not address all the barriers to multicast deployment, it revisits one key assumption at the transport layer:

> We view multicast not as a replacement for unicast, but as an optimization: Flexicast QUIC leverages multicast when available to improve the efficiency of network communication, and seamlessly falls back to unicast to ensure end-to-end connectivity.

This perspective allows us to reconsider some of the long-standing open problems that prevented large-scale deployment, even though many challenges remain. In this section, we review some of these issues and aim to provide direction for future research. We start with transport and application-oriented topics, and then focus on deployment considerations.

Heterogeneous receivers and multiple multicast flows. Correctly estimating the rate at which the source sends multicast traffic remains a difficult problem despite numerous research works on the matter during the golden age of multicast [Riz00; ML09; KB03; RFC4654; RFC5740]. Our evaluation of Flexicast QUIC considers only a single multicast flow for all receivers, which may fail to provide consistent goodput in the presence of receivers with heterogeneous network conditions as we adapt the transmission rate to the slowest receiver. Until now, we have evaded the issue by showing that the design of FCQUIC allows a server to force a single bottleneck receiver to fall back to unicast if it negatively impacts the rest of the group. Such a design raises questions, e.g., what threshold should the source use to determine whether this receiver is a bottleneck, and how/when it can rejoin the multicast flow if its condition improves. Prior work already explored receiver-driven algorithms to determine whether receivers cannot sustain the current rate [MJV96]. Such prior work could be used to extend the schedulers explored in this thesis, which focused on the liveness of the multicast flow and the evolution of the RTT variance to detect bottleneck receivers. Another strategy is to expose multiple multicast flows at different bit rates, and allow receivers to switch the flow they listen to based on their capacity [Doe2026]. Combining multiple multicast flows with potential transient unicast fallback offers possibilities that should be investigated, but require medium to large-scale experiments and involve many (heterogeneous) receivers.

Multicast flow implosion. Other reasons to increase the number of active multicast flows include receivers with different available TLS encryption algorithms or, in the case of video streaming, different video decoding schemes. Moreover, such applications expose multiple qualities (with different bit rates) depending on their clients' capacities. All these possibilities exponentially increase the potential number of active multicast flows in par-

allel. Group implosion is inherent to multicast communication, and Flexicast QUIC is no exception. Future work should investigate this question. A possibility would be to cluster receivers by their capacity and encryption algorithms, provide multicast groups only for large clusters, and leverage unicast delivery for the smallest groups.

Advanced reliability mechanism and Forward Erasure Correction. The standard retransmission mechanism of FCQUIC relies on unicast retransmission, which is not optimal in case multiple receivers experience the same loss. Still, we observed in Chapter 4 that this mechanism outperforms NORM, which relies on retransmissions and FEC in the multicast flow to recover from losses on the CloudLab testbed. This particular result stems from the testbed itself: with n receivers, a single multicast retransmission consumed n times more bandwidth than a unicast equivalent on the 100 Gbps link of the Replication Node. The design of Flexicast QUIC potentially allows extending the reliability mechanism to retransmit lost frames on the multicast flow. Future work could investigate such advanced retransmission strategies.

Forward Erasure Correction is particularly interesting for multicast communication. As presented in Section 1.8.2 (Chapter 1), a single repair packet is sufficient to recover distinct losses on different clients, thus decreasing the overall number of retransmissions performed by the source. The use of FEC has already been investigated for multicast communication [RKT98; Gem+03]. Recent works also established the benefits of FEC in QUIC [Mic+22a; MB24; Hol+25]. These modern approaches could serve as the starting point for FCQUIC, in combination with the aforementioned advanced reliability strategies.

Applications on top of FCQUIC. In Chapter 4, we leveraged HTTP/3 to transmit the manifest describing file chunking in the context of software delivery with FCQUIC, and noted that a better integration could leverage the HTTP PUSH method to fully rely on HTTP's semantics to distribute the file itself. The primary use-case of Flexicast QUIC arguably relies on large-scale video streaming. The IETF is currently standardizing WebTransport [FKV26] on top of HTTP/3, which provides a similar objective to WebSockets while leveraging QUIC's advanced semantics. In a nutshell, WebTransport provides an API for bidirectional, low-latency client-server communication, upgrading from the request/response scheme of HTTP. On top of that, Media over QUIC (MoQ) [Nan+26] is a protocol providing low-latency audio, video, and time-sensitive data on top of QUIC, and is expected to use WebTransport as an intermediary for browser integration. While WebTransport and MoQ do not significantly raise new research questions regarding multicast transport – aside from scheduling the data appropriately – integrating them with Flexicast QUIC should increase industry interest.

Flexicast QUIC for ISP deployments. This thesis focused on multicast

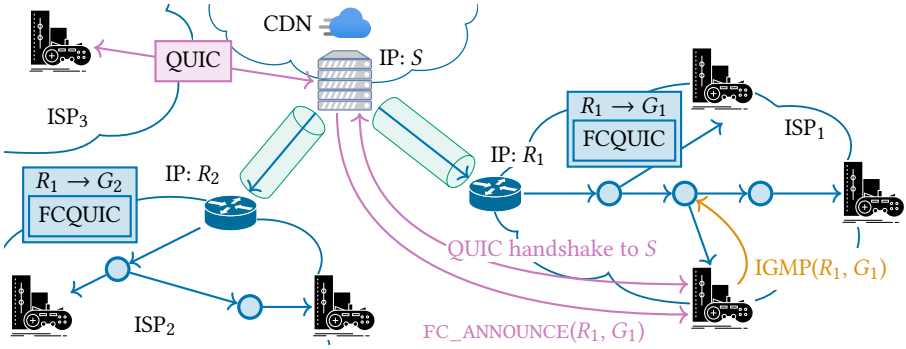


Figure 6.1: The content source (e.g., a CDN edge) sends packets through a preconfigured tunnel to the ISP relay, which is the source of the multicast tree that reaches the receivers inside this ISP. The ISP relay cannot decrypt the data exchanged over the Flexicast QUIC connection. The content source announces the multicast tree addresses (R_1, G_1) to new receivers via the `FC_ANNOUNCE` frame of FCQUIC, with R_1 the source of the tree, and G_1 the multicast group address.

challenges at the transport layer. While multicast works within domains, its primary use-cases require larger deployments, potentially at the scale of the Internet. We first focus on external entities injecting multicast packets in ISP networks, as discussed in Chapter 4 for massive software deliveries. Content Delivery Networks (CDNs) are the primary source of traffic within ISPs, but primarily use unicast to deliver this information. We briefly explain how we envision ISPs' intra-domain multicast for these CDNs and how Flexicast QUIC addresses security issues for these ISPs. However, we did not evaluate this design in this thesis. Figure 6.1 depicts this deployment architecture. This design stems from two key observations:

- Some Internet Service Providers (ISPs) deploy multicast within their domain, e.g., for IPTV [Mai+09; MM10], but are not willing to expand to inter-domain multicast for security reasons;
- Content Delivery Networks (CDNs) have dedicated peering links with ISPs, or even position servers directly within their domain.

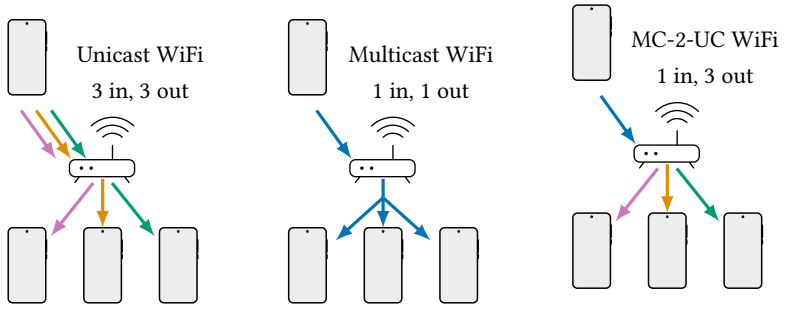
The figure shows a CDN server (S) and two participating ISPs: ISP_1 and ISP_2. Each participating ISP configures one of its routers as a multicast root ($R_{\{1,2\}}$). The source S and each ISP have contracts similar to those used by ISPs with Content Delivery Networks today. This contract specifies four main pieces of information: (i) the IP prefixes of the customers of the ISP that can be reached using multicast; (ii) the range of IP multicast addresses that the server can

use; (iii) the IP address of the multicast root responsible for this server; and (iv) the type of tunnel between the server and the router that hosts the multicast root. The multicast root can be implemented on any router. The source and multicast root could use any type of tunnel, such as GRE. When a multicast root receives a tunneled packet from a source, it simply forwards it over the multicast tree using its own IP address as the source address. For security reasons, ISPs might prefer to use a private address that is reachable only within the ISP's network as the multicast root. This is also possible with the presented design; the ISP must provide the tunnel endpoint's IP address and the multicast root's IP address to the CDN. The ISP can control the number of multicast trees that a given source can use by specifying a small block of multicast addresses to the server. From an ISP deployment perspective, this design gives ISPs complete control over the multicast trees. Each participating ISP controls which sources can send multicast packets (via tunnels) and how many multicast trees each source can create (via the allocated multicast address block). If an ISP wishes to price multicast, it can log the IGMP/MLD messages on its access routers or use NetFlow/IPFIX to measure the traffic.

To receive multicast packets, the clients of a given ISP network must know that they will be sourced by a specific multicast root, R_i, and not the source S. Using Flexicast QUIC, the source uses the FC_ANNOUNCE frame to advertise the (R_i, G_i) multicast tree to the clients served by ISP i. Upon reception of this frame, these clients send IGMP/MLD JOINs and receive the multicast packets relayed by R_i from S. It is important to note that, since Flexicast QUIC encrypts all packets, a multicast root cannot observe or modify the contents of packets sent by S. These packets remain identified by the Connection ID of the multicast flow (the Flow ID).

Automatic Multicast Tunneling and Flexicast QUIC. Automatic Multicast Tunneling (AMT) [RFC7450] enables incremental inter-domain multicast deployment, aligning with our vision with Flexicast QUIC. Known deployments of AMT provide multicast connectivity from GEANT for EUMETCast [EUMETCAST] and TreeDN [RFC9706]. Future work includes investigating how one can combine AMT with FCQUIC to deploy multicast-enabled applications on the Internet. In this thesis, we scratched the surface of this area through a demo at SIGCOMM 2025 [NB25c] as a simple proof-of-concept.

Multicast over IEEE 802.11 Wireless Media is another major theme to address when reconsidering multicast communication at a large scale. In unicast mode (Figure 6.2a), a wireless source sends each frame to the access point, which forwards them individually to each receiver. Unicast thus takes $2 \times n$ airtime slots to send the same information to n receivers. Multicast Wi-Fi theoretically offers the best scalability: a single frame is sent by the source, and the access point forwards it with a multicast MAC address, reaching all receivers in the network. However, multicast Wi-Fi (802.11) suffers



(a) Unicast. **(b) Multicast.** **(c) Multicast-to-unicast conversion.**

Figure 6.2: Multicast-to-unicast conversion offers a middle ground between pure multicast and unicast Wi-Fi. We assume that the source sends a Wi-Fi frame, i.e., it is not wired to the access point. Each color represents a distinct MAC destination address.

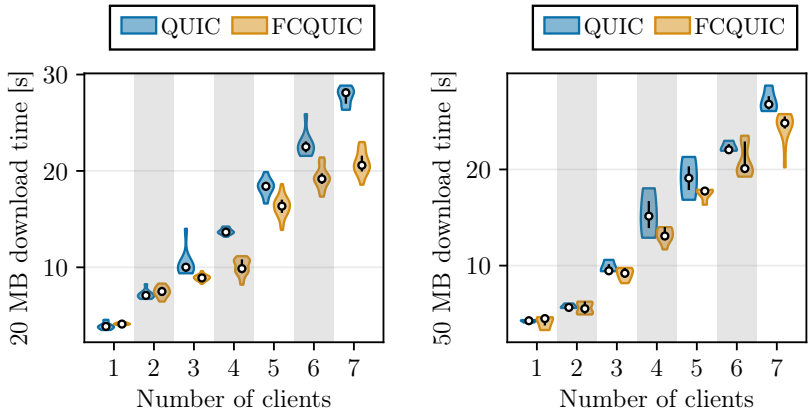
from several design issues that limit the use of multicast, even in domestic networks [RFC9119]:

1. Limited reliability: The IEEE 802.11 wireless media uses local layer-2 retransmissions in unicast communication to recover from packet losses and packet corruption. For example, we measured up to ~ 15 % of the Wi-Fi frames being lost in our testbed. This mechanism is disabled for multicast Wi-Fi, significantly increasing the observed loss rate for a reliable multicast transport protocol, e.g., FCQUIC, thereby impacting end-to-end performance.

2. Basic rate limitation: the transmission rate of the access point to send multicast frames is limited to the *basic rate*, which is usually 6 Mbps. The basic rate is the *minimum rate* at which a client can receive Wi-Fi frames.

3. Decreased overall performance: because the transmission rate is limited to the basic rate, multicast traffic requires longer airtime occupancy, thereby consuming airtime for other traffic.

4. Increased power consumption: similarly, to ensure that all receivers receive the multicast Wi-Fi frames, the access point sends them at maximum power. Second, all devices are configured to wake up on every received multicast Wi-Fi frame, thereby affecting their power-saving modes and increasing battery consumption even if they did not intend to receive the multicast traffic.



(a) 20 MB **download time in a** 2.4 GHz **802.11 network.**

(b) 50 MB **download time in a** 5 GHz **802.11 network.**

Figure 6.3: Download time of QUIC and FCQUIC in a 802.11 network using a GL-iNET AP1300LTE access point and 7 Intel NUCs as receivers. FCQUIC uses *multicast-to-unicast conversion* underneath, showing increasing gains compared to unicast delivery as the number of receivers increases, especially in the 2.4 GHz **configuration.**

The IEEE 802.11 standard includes the *multicast-to-unicast* (MC2UC) conversion mechanism, offering a middle ground between unicast and multicast Wi-Fi, as illustrated in Figure 6.2c. In this mode, the source sends a single frame, with a multicast address as the destination. The access point then replicates the frame content, targeting individual receivers who explicitly requested the content (through IGMP/MLD) using *unicast* Wi-Fi. As such, the receivers benefit from unicast Wi-Fi behavior, i.e., adapted transmission rate and local retransmissions. In this mode, n+1 airtime slots are consumed by the access point, which represents a linear increase similar to unicast mode, but reaching a ≈ 2 gain factor at scale. Multicast-to-unicast conversion is largely deployed in current access points, sometimes even completely replacing pure multicast due to its poor performance. In our demo at SIGCOMM'25 [NB25c], we demonstrated that FCQUIC over such a wireless medium in MC2UC mode could deliver a video stream with a decent quality of experience, though future work should evaluate this setup more formally.

To show the potential benefits of MC2UC in a real 802.11 wireless testbed, we briefly measure the time to transfer a file from a source to up to seven receivers on the same Wi-Fi network. Such a use-case typically occurs when multiple connected nodes need the same file as quickly as possible, e.g., speakers playing the same song synchronously. In this context, the objective is to minimize the maximum transfer time, i.e., the time at which all receivers

complete the transfer. In the unicast case, QUIC clients establish their connections and request the same file simultaneously, and the server is responsible for transmitting it to each client individually. With Flexicast QUIC, the server waits for all receivers to join the multicast flow before starting the transfer; we account for this overhead in the results. The testbed is composed of five NUCs with four Intel(R) Core(TM) i7-7567U CPUs at 3.5 GHz and three NUCs equipped with four x86 Intel(R) Pentium(R) CPUs N3700 at 1.60 GHz. We run the FCQUIC server on one of the five more recent NUCs; the seven other nodes act as clients.

Figure 6.3 reports initial results on 2.4 GHz (left) and 5 GHz (right) networks. Especially in the first case, FCQUIC provides lower download time than QUIC because the source consumes fewer airtime slots ($n + 1$ slots per packet) compared to QUIC and unicast ($2 \times n$) for n receivers. While the airtime consumption linearly increases for both methods, it is slower for MC2UC, thus leading to increasing gains when n increases. The results are less significant in the 5 GHz case with our limited testbed, since the medium offers higher available bandwidth. While these measurements are preliminary, they show that, contrary to common belief, multicast distribution in wireless media is not *worse* than unicast delivery – even better as the group size increases. However, the scale is limited due to the multicast-to-unicast conversion, which is used to benefit from efficiencies introduced for unicast wireless distribution, namely layer-2 retransmissions and higher bit rate.

We argue that these limitations once again stem from the original design idea of multicast: the source and the network must scale independently of the number of receivers; in other words, nodes should not maintain per-multicast-receiver state. Several works studied how to extend this basic multicast model to improve multicast Wi-Fi [Cha+09; Gup+16; Gup+18]. In a nutshell, these models maintain per-receiver state on the access point to enable local retransmissions and to adapt the transmission rate to the *slowest* multicast receiver's rate, rather than to a threshold based on the basic rate. However, these solutions have not been deployed in practice because they require substantial modifications to the design of current access points. We believe that more research would enhance the features available for multicast Wi-Fi. In the meantime, future work could more rigorously evaluate Flexicast QUIC in wireless 802.11 networks using the multicast-to-unicast conversion mode.

Conclusion 7

In this thesis, we explored two mechanisms to improve the efficiency of network communication, stemming from the observation that the ratio between peak and average rates is large, and continues to increase over time.

The first mechanism we proposed, HIRT, exposes Forward Erasure Correction (FEC) as a service through tunnels, transparently to end-hosts, and can protect multiple tens of gigabits of traffic per second. Such a tunnel leverages the spare bandwidth of over-provisioned networks by injecting redundant data at the tunnel ingress, allowing the egress to recover from packet losses without requiring end-hosts to entirely rely on retransmissions, which are more costly in terms of latency. We showed that, on a real network that exposes non-congestion-induced losses, HIRT reduces the latency of HTTP requests.

In the second part of this thesis, we reconsidered the use of multicast to improve the way content sources deliver identical data to numerous receivers. We focused on the transport layer, leveraging the modern features of the QUIC protocol [RFC9000] and its soon-to-be-standardized multiple paths management extension [Liu+26]. We proposed Flexicast QUIC, which exposes shared multicast flows via efficient multicast replication and per-receiver unicast paths for robustness in the event of multicast failure or underperformance of subsets of receivers. With this architecture, applications rely on a single protocol that uses multicast when available and unicast otherwise, enabling incremental deployment and avoiding the *chicken-and-egg* problem inherent to multicast communication [Dio+00]. Our design and first implementation showed that a single Flexicast QUIC source can support 1000 receivers, with a cumulative multicast traffic of 80 Gbps, on a commodity server. With Flexicast QUIC, we reconsider how we think about multicast protocols, shifting from the initial idea that multicast should scale indefinitely to focusing on improving its robustness and reliability.

This paradigm adjustment allowed us to further extend Flexicast QUIC by addressing the ACK implosion problem, which is also inherent to multicast communication. We showed that by exactly knowing the number of active receivers in a multicast group, the source can dynamically adjust an acknowledgment delay to control the rate at which receivers send feedback. Moreover, we empirically showed that this feedback rate is the primary contributor to

the CPU usage of an Flexicast QUIC source, enabling us to theoretically extrapolate our measurements based on our limited testbed.

While working on this extension, we first considered the use of negative acknowledgment to further improve scaling, which led us to analyze the optimistic acknowledgment attack in QUIC. Although QUIC's specification includes a mechanism to address this attack [RFC9000], we showed that most available QUIC implementations, including some widely used in today's Internet landscape (Cloudflare, Alibaba, Microsoft, Mozilla), were still vulnerable, and that real-world deployed servers were exposed to the attack. This work led to several patches to these vulnerable implementations.

Finally, we discussed the impact of this thesis on future research. We presented research topics that could benefit from HIRT and Flexicast QUIC as a first step. This thesis began with the idea that we should reconsider the use of multicast for large-scale one-to-many communication [NMB22]. The road to a fully working inter-domain multicast deployment at Internet scale will still be long. We presented Flexicast QUIC as a way to improve the scalability of communication for duplicate-content distribution, enabling a server to support more simultaneous clients than unicast protocols can. Another advantage of multicast communication that we did not explore in this thesis is that it uses *fewer* resources while serving the same number of clients. This aligns with the growing interest in a sustainable Internet [JV23], which aims to reduce the energy consumption of network protocols. By using network bandwidth more efficiently, Flexicast QUIC reduces the resources required to reach large numbers of receivers – a step toward the more frugal protocols that a sustainable Internet will require.

Bibliography

[CIDR2026] CIDR. *CIDR REPORT for 4 May 26*. https://www.cidr-report.org/as2.0/. 2026.

[Doe2026] A. Doeraene. "Design of a multicast congestion control algorithm for Multicast QUIC". In: ().

[EUMETCAST] R. Britton. "EumetCast Terrestrial over AMT". 2026.

[FFMPEG] "FFmpeg". In: (2023). http://www.ffmpeg.org.

[FIO] *fio*. URL: https://github.com/axboe/fio (visited on 02/15/2023).

[GSTREAMER] "GStreamer". In: (2023). https://gstreamer.freedesktop.org/.

[ITU2026] I. T. U. (ITU). *Facts and figures 2025: Internet use*. https://www.itu.int/itu-d/reports/statistics/2025/10/15/ff25-internet-use/. 2026.

[MCQUIC-IMPLEM] a. Max Franke Jake Holland. *MCQUIC implementation*. https://github.com/MaxF12/aioquic. 2023.

[MOEPGF] *MOEPGF*. URL: https://github.com/moepinet/libmoepgf.git (visited on 04/17/2022).

[NGHQ] BBC. *nghq: An implementation of Multicast QUIC*. https://github.com/bbc/nghq. 2020.

[OVH] OVH. *OVH Network Weathermap*. http://weathermap.ovh.net/. 2026.

[QUIC-APNIC] G. Huston. "A look at QUIC use". 2022.

[QUIC-LABS] A. labs. "HTTP/3 Use by Country". 2026.

[QUIC-ML-2020] Q. contributors. *QUIC Mailing list: Packet number spaces in multipath*. https://mailarchive.ietf.org/arch/msg/quic/YA7gOaKcGvdJYab_h-tawHCQfkY/. 2020.

[QUIC-W3] W. Techs. "Usage statistics of HTTP/3 for websites". 2026.

[QUICBLOG] J. Roskind. "Experimenting with QUIC". 2013.

[QUICCHROMIUM] R. Hamilton. "Issue 11125002: Add QuicFramer and friends." 2012.

[QUICHE] Cloudflare. *Quiche: savoury implementation of the QUIC transport protocol and HTTP/3.* https://github.com/cloudflare/quiche. 2023.

[QUICHE-MULTIPATH] Q. D. Coninck and Cloudflare. *Multipath support with non-zero length Connection IDs.* https://github.com/cloudflare/quiche/pull/1310. 2022.

[QUICWG] IETF. "IETF QUIC Working Group". 2016.

[RFC1112] D. S. E. Deering. *Host extensions for IP multicasting.* RFC 1112. Aug. 1989. DOI: 10.17487/RFC1112.

[RFC1122] R. T. Braden. *Requirements for Internet Hosts - Communication Layers.* RFC 1122. Oct. 1989. DOI: 10.17487/RFC1122.

[RFC1195] *Use of OSI IS-IS for routing in TCP/IP and dual environments.* RFC 1195. Dec. 1990. DOI: 10.17487/RFC1195.

[RFC2018] S. Floyd, J. Mahdavi, M. Mathis, and D. A. Romanow. *TCP Selective Acknowledgment Options.* RFC 2018. Oct. 1996. DOI: 10.17487/RFC2018.

[RFC2236] B. Fenner. *Internet Group Management Protocol, Version 2.* RFC 2236. Nov. 1997. DOI: 10.17487/RFC2236.

[RFC2246] C. Allen and T. Dierks. *The TLS Protocol Version 1.0.* RFC 2246. Jan. 1999. DOI: 10.17487/RFC2246.

[RFC2616] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. June 1999. DOI: 10.17487/RFC2616.

[RFC2710] D. S. E. Deering, B. Fenner, and B. Haberman. *Multicast Listener Discovery (MLD) for IPv6.* RFC 2710. Oct. 1999. DOI: 10.17487/RFC2710.

[RFC2988] D. V. Paxson and M. Allman. *Computing TCP's Retransmission Timer.* RFC 2988. Nov. 2000. DOI: 10.17487/RFC2988.

[RFC3031] A. Viswanathan, E. C. Rosen, and R. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031. Jan. 2001. DOI: 10.17487/RFC3031.

[RFC3168] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: 10.17487/RFC3168.

[RFC3376] B. Cain, D. S. E. Deering, B. Fenner, I. Kouvelas, and A. Thyagarajan. *Internet Group Management Protocol, Version 3*. RFC 3376. Oct. 2002. DOI: 10.17487/RFC3376.

[RFC3550] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. July 2003. DOI: 10.17487/RFC3550.

[RFC3810] L. Costa and R. Vida. *Multicast Listener Discovery Version 2 (MLDv2) for IPv6*. RFC 3810. June 2004. DOI: 10.17487/RFC3810.

[RFC3973] J. Nicholas, A. Adams, and W. Siadak. *Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised)*. RFC 3973. Jan. 2005. DOI: 10.17487/RFC3973.

[RFC4253] C. M. Lonvick and T. Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. Jan. 2006. DOI: 10.17487/RFC4253.

[RFC4271] Y. Rekhter, S. Hares, and T. Li. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. Jan. 2006. DOI: 10.17487/RFC4271.

[RFC4607] H. Holbrook and B. Cain. *Source-Specific Multicast for IP*. RFC 4607. Aug. 2006. DOI: 10.17487/RFC4607.

[RFC4654] M. J. Handley and J. Widmer. *TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification*. RFC 4654. Aug. 2006. DOI: 10.17487/RFC4654.

[RFC4760] R. Chandra, T. J. Bates, Y. Rekhter, and D. Katz. *Multiprotocol Extensions for BGP-4*. RFC 4760. Jan. 2007. DOI: 10.17487/RFC4760.

[RFC4960] R. R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. Sept. 2007. DOI: 10.17487/RFC4960.

[RFC5340] D. Ferguson, A. Lindem, and J. Moy. *OSPF for IPv6*. RFC 5340. July 2008. DOI: 10.17487/RFC5340.

[RFC5681] E. Blanton, D. V. Paxson, and M. Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681.

[RFC5740] J. P. Macker, C. Bormann, M. J. Handley, and B. Adamson. *NACK-Oriented Reliable Multicast (NORM) Transport Protocol*. RFC 5740. Nov. 2009. DOI: 10.17487/RFC5740.

[RFC5775] L. Vicisano, M. Watson, and M. Luby. *Asynchronous Layered Coding (ALC) Protocol Instantiation*. RFC 5775. Apr. 2010. DOI: 10.17487/RFC5775.

[RFC5869] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869.

[RFC6298] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. *Computing TCP's Retransmission Timer*. RFC 6298. June 2011. DOI: 10.17487/RFC6298.

[RFC6356] C. Raiciu, M. J. Handley, and D. Wischik. *Coupled Congestion Control for Multipath Transport Protocols*. RFC 6356. Oct. 2011. DOI: 10.17487/RFC6356.

[RFC6675] E. Blanton, M. Allman, L. Wang, I. Järvinen, M. Kojo, and Y. Nishida. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. RFC 6675. Aug. 2012. DOI: 10.17487/RFC6675.

[RFC6726] T. Paila, R. Walsh, M. Luby, V. Roca, and R. Lehtonen. *FLUTE - File Delivery over Unidirectional Transport*. RFC 6726. Nov. 2012. DOI: 10.17487/RFC6726.

[RFC7046] M. Wählisch, T. C. Schmidt, and S. Venaas. *A Common API for Transparent Hybrid Multicast*. RFC 7046. Dec. 2013. DOI: 10.17487/RFC7046.

[RFC7323] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger. *TCP Extensions for High Performance*. RFC 7323. Sept. 2014. DOI: 10.17487/RFC7323.

[RFC7450] G. Bumgardner. *Automatic Multicast Tunneling*. RFC 7450. Feb. 2015. DOI: 10.17487/RFC7450.

[RFC768] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[RFC7761] B. Fenner, M. J. Handley, H. Holbrook, I. Kouvelas, R. Parekh, Z. (Zhang, and L. Zheng. *Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)*. RFC 7761. Mar. 2016. DOI: 10.17487/RFC7761.

[RFC791] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791.

[RFC8200] D. S. E. Deering and B. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. July 2017. DOI: 10.17487/RFC8200.

[RFC8279] I. Wijnands, E. C. Rosen, A. Dolganow, T. Przygienda, and S. Aldrin. *Multicast Using Bit Index Explicit Replication (BIER)*. RFC 8279. Nov. 2017. DOI: 10.17487/RFC8279.

[RFC8402] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. *Segment Routing Architecture*. RFC 8402. July 2018. DOI: 10.17487/RFC8402.

[RFC8446] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446.

[RFC8681] V. Roca and B. Teibi. *Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME*. RFC 8681. Jan. 2020. DOI: 10.17487/RFC8681.

[RFC8684] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 8684. Mar. 2020. DOI: 10.17487/RFC8684.

[RFC8754] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer. *IPv6 Segment Routing Header (SRH)*. RFC 8754. Mar. 2020. DOI: 10.17487/RFC8754.

[RFC8777] J. Holland. *DNS Reverse IP Automatic Multicast Tunneling (AMT) Discovery*. RFC 8777. Apr. 2020. DOI: 10.17487/RFC8777.

[RFC8815] M. Abrahamsson, T. Chown, L. Giuliano, and T. Eckert. *Deprecating Any-Source Multicast (ASM) for Interdomain Multicast*. RFC 8815. Aug. 2020. DOI: 10.17487/RFC8815.

[RFC8985] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha. *The RACK-TLP Loss Detection Algorithm for TCP*. RFC 8985. Feb. 2021. DOI: 10.17487/RFC8985.

[RFC8986] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li. *Segment Routing over IPv6 (SRv6) Network Programming*. RFC 8986. Feb. 2021. DOI: 10.17487/RFC8986.

[RFC9000] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000.

[RFC9001] M. Thomson and S. Turner. *Using TLS to Secure QUIC*. RFC 9001. May 2021. DOI: 10.17487/RFC9001.

[RFC9002] J. Iyengar and I. Swett. *QUIC Loss Detection and Congestion Control*. RFC 9002. May 2021. DOI: 10.17487/RFC9002.

[RFC9113] M. Thomson and C. Benfield. *HTTP/2*. RFC 9113. June 2022. DOI: 10.17487/RFC9113.

[RFC9114] M. Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114.

[RFC9119] C. E. Perkins, M. McBride, D. Stanley, W. Kumari, and J.-C. Zúñiga. *Multicast Considerations over IEEE 802 Wireless Media*. RFC 9119. Oct. 2021. DOI: 10.17487/RFC9119.

[RFC9221] T. Pauly, E. Kinnear, and D. Schinazi. *An Unreliable Datagram Extension to QUIC*. RFC 9221. Mar. 2022. DOI: 10.17487/RFC9221.

[RFC9250] C. Huitema, S. Dickinson, and A. Mankin. *DNS over Dedicated QUIC Connections*. RFC 9250. May 2022. DOI: 10.17487/RFC9250.

[RFC9260] R. R. Stewart, M. Tüxen, and karen Nielsen. *Stream Control Transmission Protocol*. RFC 9260. June 2022. DOI: 10.17487/RFC9260.

[RFC9262] T. Eckert, M. Menth, and G. Cauchie. *Tree Engineering for Bit Index Explicit Replication (BIER-TE)*. RFC 9262. Oct. 2022. DOI: 10.17487/RFC9262.

[RFC9265] N. Kuhn, E. Lochin, F. Michel, and M. Welzl. *Forward Erasure Correction (FEC) Coding and Congestion Control in Transport*. RFC 9265. July 2022. DOI: 10.17487/RFC9265.

[RFC9293] W. Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293.

[RFC9298] D. Schinazi. *Proxying UDP in HTTP*. RFC 9298. Aug. 2022. DOI: 10.17487/RFC9298.

[RFC9330] B. Briscoe, K. D. Schepper, M. Bagnulo, and G. White. *Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture*. RFC 9330. Jan. 2023. DOI: 10.17487/RFC9330.

[RFC9438] L. Xu, S. Ha, I. Rhee, V. Goel, and L. Eggert. *CUBIC for Fast and Long-Distance Networks*. RFC 9438. Aug. 2023. DOI: 10.17487/RFC9438.

[RFC9706] L. Giuliano, C. Lenart, and R. Adam. *TreeDN: Tree-Based Content Delivery Network (CDN) for Live Streaming to Mass Audiences*. RFC 9706. Jan. 2025. DOI: 10.17487/RFC9706.

[SOCKET] L. manual page. "Socket(2)". 2026.

[STARLINK] *Starlink*. 2023.

[TOKIO] Tokio. *Tokio: An asynchronous Rust runtime*. https://tokio.rs. 2024.

[WRK] *wrk*. URL: https://github.com/wg/wrk (visited on 04/19/2022).

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

[23] *picoquic*. 2023.

[24] "YouTube recommended upload encoding settings". In: (2024). https://support.google.com/youtube/answer/1722171?hl=en#zippy=%2Cvideo-codec-h%2Ccontainer-mp%2Caudio-codec-aac-lc%2Cbitrate.

[AHH05] J.-S. Ahn, S.-W. Hong, and J. Heidemann. "An adaptive FEC code control algorithm for mobile wireless sensor networks". In: *Journal of Communications and Networks* 7.4 (2005), pp. 489–498.

[AJ 25] O. G. AJ Curchill. "Discover PlayStation Game Install Sizes: A Reference Guide". 2025.

[Aka] Akamai. *Akamai Online Retail Performance Report: Milliseconds Are Critical*. https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp.

[Aka22] Akamai. "Oops, we did it again". In: (2022). https://www.linkedin.com/pulse/oops-we-did-again-akamai-technologies.

[Aka26] Akamai. "Akamai Global Infrastructure". 2026.

[AM02] R. B. Adamson and J. P. Macker. "Quantitative prediction of NACK-oriented reliable multicast (NORM) feedback". In: *MILCOM 2002. Proceedings*. Vol. 2. IEEE. 2002, pp. 964–969.

[Arm00] G. Armitage. "MPLS: the magic behind the myths [multiprotocol label switching]". In: *IEEE Communications Magazine* 38.1 (2000), pp. 124–131.

[ASG21] S. Abdous, E. Sharafzadeh, and S. Ghorbani. "Burst-tolerant datacenter networks with Vertigo". In: *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 2021, pp. 1–15.

[Aub+16] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure. "SCMon: Leveraging segment routing to improve network monitoring". In: *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE. 2016, pp. 1–9.

[Bad+17] A. Badr, A. Khisti, W.-T. Tan, and J. Apostolopoulos. "Perfecting protection for interactive multimedia: A survey of forward error correction for low-delay interactive applications". In: *IEEE Signal Processing Magazine* 34.2 (2017), pp. 95–113.

[Bal+11] M. Balakrishnan, T. Marian, K. P. Birman, H. Weatherspoon, and L. Ganesh. "Maelstrom: Transparent Error Correction for Communication Between Data Centers". In: *IEEE/ACM Transactions on Networking* 19.3 (June 2011), pp. 617–629. ISSN: 1063-6692, 1558-2566. DOI: 10.1109/TNET.2011.2144616. (Visited on 01/11/2022).

[Bal+22] A. Baltaci, H. Cech, N. Mohan, F. Geyer, V. Bajpai, J. Ott, and D. Schupke. "Analyzing real-time video delivery over cellular networks for remote piloting aerial vehicles". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 98–112.

[BAM10] T. Benson, A. Akella, and D. A. Maltz. "Network traffic characteristics of data centers in the wild". In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 2010, pp. 267–280.

[Bar+19] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić. "RSS++ load and state-aware receive side scaling". In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 2019, pp. 318–333.

[Bar+21] T. Barbette, E. Wu, D. Kostić, G. Q. Maguire, P. Papadimitratos, and M. Chiesa. "Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency". In: *IEEE/ACM Transactions on Networking* 30.1 (2021), pp. 354–367.

[Bar24] T. Barbette. "Poster: NPF: orchestrate and reproduce network experiments". In: *2024 ACM Conference on Reproducibility and Replicability*. 2024.

[BCZ97] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. "An architecture for active networking". In: *International Conference on High Performance Networking*. Springer. 1997, pp. 265–279.

[Ber06] T. Berger. "Analysis of current VPN technologies". In: *First International Conference on Availability, Reliability and Security (ARES'06)*. IEEE. 2006, 8–pp.

[Ber25] R. van der Berg. "Is 8 terabit per second too much traffic?" 2025.

[BFT99] J.-C. Bolot, S. Fosse-Parisis, and D. Towsley. "Adaptive FEC-based error control for Internet telephony". In: *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*. Vol. 3. IEEE. 1999, pp. 1453–1460.

[Ble+18] 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.

[BLM02] J. W. Byers, M. Luby, and M. Mitzenmacher. "A digital fountain approach to asynchronous reliable multicast". In: *IEEE Journal on Selected areas in Communications* 20.8 (2002), pp. 1528–1540.

[Bos+14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[BP25] A. Buchet and C. Pelsser. "An Analysis of QUIC Connection Migration in the Wild". In: *ACM SIGCOMM Computer Communication Review* 55.1 (2025), pp. 3–9.

[BS02] J. Bellardo and S. Savage. "Measuring packet reordering". In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. 2002, pp. 97–105.

[BSM15] T. Barbette, C. Soldani, and L. Mathy. "Fast userspace packet processing". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2015, pp. 5–16.

[BWL07] J. Bi, J. Wu, and X. Leng. "IPv4/IPv6 transition technologies and univer6 architecture". In: *International Journal of Computer Science and Network Security* 7.1 (2007), pp. 232–243.

[Bye+98] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. "A digital fountain approach to reliable distribution of bulk data". In: *ACM SIGCOMM Computer Communication Review* 28.4 (1998), pp. 56–67.

[Car+16] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. "Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time". In: *Queue* 14.5 (2016), pp. 20–53.

[Car+17] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. "BBR: Congestion-based congestion control". In: *Communications of the ACM* 60.2 (2017), pp. 58–66.

[Cha+09] R. Chandra, S. Karanth, T. Moscibroda, V. Navda, J. Padhye, R. Ramjee, and L. Ravindranath. "DirCast: A practical and efficient Wi-Fi multicast system". In: *2009 17th IEEE International Conference on Network Protocols*. IEEE. 2009, pp. 161–170.

[Che+09] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. "Understanding TCP incast throughput collapse in datacenter networks". In: *Proceedings of the 1st ACM workshop on Research on enterprise networking*. 2009, pp. 73–82.

[Che+14] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. *TCP Fast Open*. RFC 7413. Dec. 2014. DOI: 10.17487/RFC7413.

[cisco24] *A New Era of Connectivity: Cisco SD-WAN and Starlink Transforming Networks*. URL: https://learningnetwork.cisco.com/s/article/a-new-era-of-connectivity-cisco-sd-wan-and-starlink-transforming-networks (visited on 01/12/2026).

[Cla+17] 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". In: *The Computer Games Journal* 6.1 (2017), pp. 33–61.

[Coh+20a] A. Cohen, D. Malak, V. B. Bracha, and M. Médard. "Adaptive causal network coding with feedback". In: *IEEE Transactions on Communications* 68.7 (2020), pp. 4325–4341.

[Coh+20b] A. Cohen, G. Thiran, V. B. Bracha, and M. Médard. "Adaptive causal network coding with feedback for multipath multi-hop communications". In: *IEEE Transactions on Communications* 69.2 (2020), pp. 766–785.

[COM10] F. Chang, K. Onohara, and T. Mizuochi. "Forward error correction for 100 G transport networks". In: *IEEE Communications Magazine* 48.3 (2010), S48–S55.

[CRZ00] Y.-h. Chu, S. G. Rao, and H. Zhang. "A case for end system multicast (keynote address)". In: *ACM SIGMETRICS Performance Evaluation Review* 28.1 (2000), pp. 1–12.

[CSB25] N. Cardwell, I. Swett, and J. Beshay. *BBR Congestion Control*. Internet-Draft draft-ietf-ccwg-bbr-04. Work in Progress. Internet Engineering Task Force, Oct. 2025. 81 pp.

[CSB26] N. Cardwell, I. Swett, and J. Beshay. *BBR Congestion Control*. Internet-Draft draft-ietf-ccwg-bbr-05. Work in Progress. Internet Engineering Task Force, Mar. 2026. 84 pp.

[CZT99] A. Chockalingam, M. Zorzi, and V. Tralli. "Wireless TCP performance with link layer FEC/ARQ". In: *1999 IEEE International Conference on Communications (Cat. No. 99CH36311)*. Vol. 2. IEEE. 1999, pp. 1212–1216.

[DB17] Q. De Coninck and O. Bonaventure. "Multipath quic: Design and evaluation". In: *Proceedings of the 13th international conference on emerging networking experiments and technologies*. 2017, pp. 160–166.

[DB21] Q. De Coninck and O. Bonaventure. "Multiflow QUIC: A generic multipath transport protocol". In: *IEEE Communications Magazine* 59.5 (2021).

[DBH25] M. Duke, N. Banks, and C. Huitema. *QUIC-LB: Generating Routable QUIC Connection IDs*. Internet-Draft draft-ietf-quic-load-balancers-21. Work in Progress. Internet Engineering Task Force, Aug. 2025. 45 pp.

[De +19] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure. "Pluginizing quic". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 59–74.

[De 22] Q. De Coninck. "The packet number space debate in multipath QUIC". In: *ACM SIGCOMM Computer Communication Review* 52.3 (2022), pp. 2–9.

[Dee88] S. E. Deering. “Multicast routing in internetworks and extended LANs”. In: *Symposium proceedings on Communications architectures and protocols*. 1988, pp. 55–64.

[Det+15] J. Detchart, E. Lochin, J. Lacan, and V. Roca. “Tetrys, an on-the-fly network coding protocol”. In: (2015).

[Dio+00] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. “Deployment issues for the IP multicast service and architecture”. In: *IEEE network* 14.1 (2000), pp. 78–88.

[Don17] J. A. Donenfeld. “Wireguard: next generation kernel network tunnel.” In: *NDSS*. 2017, pp. 1–12.

[Dub+17] E. Dubois, N. Kuhn, J.-B. Dupé, P. Gélard, F. Arnal, C. Baudoin, A. Delrieu, and D. Pradas. “OpenSAND, an open source satcom Emulator”. In: *Proc. Kaconf*. 2017, pp. 1–4.

[Dup+19] 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.

[Dut24] C. of Duty Staff. “The Road to Black Ops 6 Launch: Preload and New UI”. 2024.

[Eri94] H. Eriksson. “Mbone: The multicast backbone”. In: *Communications of the ACM* 37.8 (1994), pp. 54–60.

[Fab98] T. Faber. “ACC: using active networking to enhance feedback congestion control mechanisms”. In: *IEEE network* 12.3 (1998), pp. 61–65.

[FF96] K. Fall and S. Floyd. “Simulation-based comparisons of Tahoe, Reno and SACK TCP”. In: *ACM SIGCOMM Computer Communication Review* 26.3 (1996), pp. 5–21.

[FHS24] 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.

[Fil+15] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois. "The segment routing architecture". In: *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2015, pp. 1–6.

[FKV26] A. Frindell, E. Kinnear, and V. Vasiliev. *WebTransport over HTTP/3*. Internet-Draft draft-ietf-webtrans-http3-15. Work in Progress. Internet Engineering Task Force, Mar. 2026. 40 pp.

[Fou15] L. Foundation. *Data Plane Development Kit (DPDK)*. 2015.

[Fuk+10] K. Fukuchi, D. Ogasahara, J. Hu, T. Takamichi, T. Koga, M. Sato, E. de Gabory, Y. Hashimoto, T. Yoshihara, W. Maeda, et al. "112Gb/s optical transponder with PM-QPSK and coherent detection employing parallel FPGA-based real-time digital signal processing, FEC and 100GbE Ethernet interface". In: *36th European Conference and Exhibition on Optical Communication*. IEEE. 2010, pp. 1–3.

[Gem+03] J. Gemmell, T. Montgomery, T. Speakman, and J. Crowcroft. "The PGM reliable multicast protocol". In: *IEEE network* 17.1 (2003), pp. 16–22.

[GHK02] R.-H. Gau, Z. J. Haas, and B. Krishnamachari. "On multicast flow control for heterogeneous receivers". In: *IEEE/ACM Transactions on Networking* 10.1 (2002), pp. 86–101.

[GMP96] 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.

[GN12] J. Gettys and K. Nichols. "Bufferbloat: dark buffers in the internet". In: *Communications of the ACM* 55.1 (2012), pp. 57–65.

[Gup+16] V. Gupta, Y. Bejerano, C. Gutterman, J. Ferragut, K. Guo, T. Nandagopal, and G. Zussman. "Light-weight feedback mechanism for wifi multicast to very large groups—experimental evaluation". In: *IEEE/ACM Transactions on Networking* 24.6 (2016), pp. 3826–3840.

[Gup+18] V. Gupta, C. Gutterman, Y. Bejerano, and G. Zussman. "Experimental evaluation of large scale WiFi multicast rate control". In: *IEEE Transactions on Wireless Communications* 17.4 (2018), pp. 2319–2332.

[GWP99] 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.

[HD15] O. Haq and F. R. Dogar. "Leveraging the Power of Cloud for Reliable Wide Area Communication". In: *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. HotNets-XIV. Philadelphia, PA, USA: Association for Computing Machinery, 2015. ISBN: 9781450340472. DOI: 10.1145/2834050.2834109.

[HH08] G. Haßlinger and O. Hohlfeld. "The Gilbert-Elliott model for packet loss in real time services on the Internet". In: *14th GI/ITG Conference-Measurement, Modelling and Evalutation of Computer and Communication Systems*. VDE. 2008, pp. 1–15.

[Hol+25] K. Holzinger, D. Petri, S. Lachnit, M. Kempf, H. Stubbe, S. Gallenmüller, S. Günther, and G. Carle. "Forward error correction and weighted hierarchical fair multiplexing for http/3 over quic". In: *Proceedings of the IEEE* (2025), pp. 1–9.

[Hol+26] J. Holland, L. Pardue, M. Franke, and K. Rose. *Multicast Extension for QUIC*. Internet-Draft draft-jholland-quic-multicast-08. Work in Progress. Internet Engineering Task Force, Jan. 2026. 40 pp.

[Hol20] J. Holland. "IP Multicast: Next steps to make it real". NANOG79, available from `https://youtu.be/2aihLUb1elg?list=PLO8DR5ZGla8i-aVXtTFRZ6l7BRBvYdrkO`. 2020.

[Hon+11] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. "Is it still possible to extend TCP?" In: *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. 2011, pp. 181–194.

[HRD17] O. Haq, M. Raja, and F. R. Dogar. "Measuring and Improving the Reliability of Wide-Area Cloud Paths". In: *Proceedings of the 26th International Conference on World Wide Web*. WWW '17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 253–262. ISBN: 9781450349130. DOI: 10.1145/3038912.3052560.

[HRF25] J. Holland, K. Rose, and M. Franke. *Asymmetric Manifest Based Integrity*. Internet-Draft draft-ietf-mboned-ambi-05. Work in Progress. Internet Engineering Task Force, Oct. 2025. 29 pp.

[Hui96] C. Huitema. "The case for packet level FEC". In: *International Workshop on Protocols for High Speed Networks*. Springer. 1996, pp. 109–120.

[Hus25] G. Huston. *A Day in the Life of BGP*. https://www.potaroo.net/ispcol/2025-06/daybgp.htm l. 2025.

[ID25] J. Iurman and B. Donnet. "The Razor's Edge: IPv6 Extension Headers Survivability". In: *Passive and Active Measurement: 26th International Conference, PAM 2025, Virtual Event, March 10–12, 2025, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2025, pp. 3–29. ISBN: 978-3-031-85959-5. DOI: 10.1007/978-3-031-85960-1_1.

[IET25] Q. working group (IETF). *QUIC Implementations*. Accessed on April 4, 2025. 2025.

[ISK25] J. Iyengar, I. Swett, and M. Kühlewind. *QUIC Acknowledgment Frequency*. Internet-Draft draft-ietf-quic-ack-frequency-13. Work in Progress. Internet Engineering Task Force, Nov. 2025. 19 pp.

[Jac88] V. Jacobson. "Congestion avoidance and control". In: *ACM SIGCOMM computer communication review* 18.4 (1988), pp. 314–329.

[Jad+19] M. Jadin, F. Aubry, P. Schaus, and O. Bonaventure. "CG4SR: Near optimal traffic engineering for segment routing with column generation". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1333–1341.

[Jae+23] 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.

[Jak] G. Jake Brutlag. *Speed Matters*.

[Jan+00] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole Jr. "Overcast: Reliable multicasting with an overlay network". In: *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*. 2000.

[JV23] R. Jacob and L. Vanbever. "The internet of tomorrow must sleep more and grow old". In: *ACM SIGENERGY Energy Informatics Review* 3.3 (2023), pp. 27–32.

[Kam+24] V. Kamel, J. Zhao, D. Li, and J. Pan. "StarQUIC: Tuning congestion control algorithms for QUIC over LEO satellite networks". In: *Proceedings of the 2nd International Workshop on LEO Networking and Communication*. 2024, pp. 43–48.

[Kas+22] M. M. Kassem, A. Raman, D. Perino, and N. Sastry. "A browser-side view of starlink connectivity". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 151–158.

[KB03] G.-I. Kwon and J. W. Byers. "Smooth multirate multicast congestion control". In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*. Vol. 2. IEEE. 2003, pp. 1022–1032.

[Kem+24] M. Kempf, B. Jaeger, J. Zirngibl, K. Ploch, and G. Carle. "QUIC on the Fast Lane: Extending Performance Evaluations on High-rate Links". In: *Computer Communications* 223 (2024), pp. 90–100.

[Kim+12] M. Kim, J. Cloud, A. ParandehGheibi, L. Urbina, K. Fouli, D. Leith, and M. Médard. "Network coded tcp (ctcp)". In: *arXiv preprint arXiv:1212.2291* (2012).

[Kir12] R. E. Kirk. *Experimental design: Procedures for the behavioral sciences*. Sage Publications, 2012.

[Koh+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[Lab25] U. N. R. Laboratory. *NACK-Oriented Reliable Multicast (NORM) implementation and tools (RFCs 5740, 5401)*. `https://github.com/USNavalResearchLaboratory/norm`. 2025.

[Lai+25] Z. Lai, Z. Li, Q. Wu, H. Li, J. Li, X. Xie, Y. Li, J. Liu, and J. Wu. "Leocc: Making internet congestion control robust to leo satellite dynamics". In: *Proceedings of the ACM SIGCOMM 2025 Conference*. 2025, pp. 129–146.

[Lan+17] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. "The quic transport protocol: Design and internet-scale deployment". In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196.

[Lar+21] A. Laraba, J. François, S. R. Chowdhury, I. Chrisment, and R. Boutaba. "Mitigating TCP protocol misuse with programmable data planes". In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 760–774.

[LB17] D. Lebrun and O. Bonaventure. "Implementing ipv6 segment routing in the linux kernel". In: *Proceedings of the Applied Networking Research Workshop*. 2017, pp. 35–41.

[LG96] 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.

[Li+22] J. Li, Z. Li, R. Lu, K. Xiao, S. Li, J. Chen, J. Yang, C. Zong, A. Chen, Q. Wu, et al. "Livenet: a low-latency video transport network for large-scale live streaming". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 812–825.

[Lin06] Linux. "[TCP]: make cubic the default". 2006.

[Liu+24] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind. *Multipath Extension for QUIC*. Internet-Draft draft-ietf-quic-multipath-11. Work in Progress. Internet Engineering Task Force, Oct. 2024. 38 pp.

[Liu+26] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind. *Managing multiple paths for a QUIC connection*. Internet-Draft draft-ietf-quic-multipath-19. Work in Progress. Internet Engineering Task Force, Jan. 2026. 38 pp.

[LK04] H. Lundqvist and G. Karlsson. "TCP with end-to-end FEC". In: *International Zurich Seminar on Communications, 2004*. IEEE. 2004, pp. 152–155.

[LP96] J. C. Lin and S. Paul. "RMTP: A reliable multicast transport protocol". In: *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*. Vol. 3. IEEE. 1996, pp. 1414–1424.

[Lub+02] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft. *The use of forward error correction (FEC) in reliable multicast*. Tech. rep. RFC 3453, December, 2002.

[Ma+23] S. Ma, Y. C. Chou, H. Zhao, L. Chen, X. Ma, and J. Liu. "Network characteristics of LEO satellite constellations: A Starlink-based measurement from end users". In: *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE. 2023, pp. 1–10.

[Mai+09] J. Maisonneuve, M. Deschanel, J. Heiles, W. Li, H. Liu, R. Sharpe, and Y. Wu. "An overview of IPTV standards development". In: *IEEE Transactions on broadcasting* 55.2 (2009), pp. 315–328.

[MB24] F. Michel and O. Bonaventure. "QUIRL: Flexible QUIC Loss Recovery for Low Latency Applications". In: *IEEE/ACM Transactions on Networking* (2024).

[MDB19] F. Michel, Q. De Coninck, and O. Bonaventure. "QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC". In: *2019 IFIP Networking Conference (IFIP Networking)*. IEEE. 2019, pp. 1–9.

[Mic+22a] F. Michel, A. Cohen, D. Malak, Q. De Coninck, M. Médard, and O. Bonaventure. "FlEC: Enhancing QUIC with application-tailored reliability mechanisms". In: *IEEE/ACM Transactions on Networking* (2022).

[Mic+22b] F. Michel, M. Trevisan, D. Giordano, and O. Bonaventure. "A first look at starlink performance". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 130–136.

[Mic22] Microsoft. "Use multicast to deploy Windows over the network with Configuration Manager". 2022.

[Mic23] F. Michel. "Flexible QUIC loss recovery mechanisms for latencysensitive applications". In: *Bonaventure, Olivier* (2023).

[Mis+19] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi, and B. Leong. "The great internet TCP congestion control census". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3.3 (2019), pp. 1–24.

[MJ+93] S. McCanne, V. Jacobson, et al. "The BSD packet filter: A new architecture for user-level packet capture." In: *USENIX winter*. Vol. 46. 1993, pp. 259–270.

[MJ95] S. McCanne and V. Jacobson. "vic: A flexible framework for packet video". In: *Proceedings of the third ACM international conference on Multimedia*. 1995, pp. 511–522.

[MJV96] S. McCanne, V. Jacobson, and M. Vetterli. "Receiver-driven layered multicast". In: *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*. 1996, pp. 117–130.

[MK26] J.-L. Menne and M. Kühlewind. "Overview of Stream-Aware Scheduling for QUIC-Based Proxying Using Multiple Paths". In: *IEEE Communications Standards Magazine* (2026).

[ML09] A. Matrawy and I. Lambadaris. "A survey of congestion control schemes for multicast video applications". In: *IEEE Communications Surveys & Tutorials* 5.2 (2009), pp. 22–31.

[MM10] M. Mellia and M. Meo. “Measurement of IPTV traffic from an operative network”. In: *European Transactions on Telecommunications* 21.4 (2010), pp. 324–336.

[Nam25a] F. L. (Namex). “Have you ever seen the shape of an Apple ios update?” 2025.

[Nam25b] F. L. (Namex). “Namex Steps Into the Terabit Era”. 2025.

[Nam25c] F. L. (Namex). “On the occasion of Microsoft Patch Tuesday”. 2025.

[Nam25d] F. L. (Namex). “Yesterday Fortnite launched a massive update that significantly impacted the network, visible also through IXPs”. 2025.

[Nan+26] S. Nandakumar, V. Vasiliev, I. Swett, and A. Frindell. *Media over QUIC Transport*. Internet-Draft draft-ietf-moq-transport-17. Work in Progress. Internet Engineering Task Force, Mar. 2026. 121 pp.

[Nav25] L. Navarre. *Flexicast QUIC Deployment at ACM SIGCOMM 2025 demo*. `https://github.com/louisna/fcquic-sigcomm-demo-2025`. 2025.

[Nav26a] L. Navarre. *Experiment scripts for Scaling Flexicast QUIC for Large-Scale Software Distribution*. `https://github.com/louisna/fcquic-ack-experiments-ToN`. 2026.

[Nav26b] L. Navarre. *Flexicast QUIC implementation based on Cloudflare quiche*. `https://github.com/IPNetworkingLab/flexicast-quic`. 2026.

[NB25a] L. Navarre and O. Bonaventure. *Flexicast QUIC: combining unicast and multicast in a single QUIC connection*. Internet-Draft draft-navarre-quic-flexicast-01. Work in Progress. Internet Engineering Task Force, July 2025. 27 pp.

[NPB23] L. Navarre, O. Pereira, and O. Bonaventure. “MC-QUIC: Multicast and unicast in a single transport protocol”. In: *arXiv preprint arXiv:2309.06633* (2023).

[Obr02] K. Obraczka. “Multicast transport protocols: a survey and taxonomy”. In: *IEEE Communications magazine* 36.1 (2002), pp. 94–102.

[ope25] openreach. “Fortnite patches boost record-breaking UK broadband traffic”. 2025.

[Paa+14] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. "Experimental evaluation of multipath TCP schedulers". In: *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*. 2014, pp. 27–32.

[Pau+97] S. Paul, K. K. Sabnani, J.-H. Lin, and S. Bhattacharyya. "Reliable multicast transport protocol (RMTP)". In: *IEEE journal on selected areas in communications* 15.3 (1997), pp. 407–421.

[PB20] M. Piraux and O. Bonaventure. *Tunneling Internet protocols inside QUIC*. Tech. rep. Internet-Draft draft-piraux-quic-tunnel-01. IETF Secretariat. https://tools …, 2020.

[PBH22] L. Pardue, R. Bradbury, and S. Hurst. *Hypertext Transfer Protocol (HTTP) over multicast QUIC*. Internet-Draft draft-pardue-quic-http-mcast-11. Work in Progress. Internet Engineering Task Force, July 2022. 68 pp.

[Per+22] D. Perdices, G. Perna, M. Trevisan, D. Giordano, and M. Mellia. "When satellite is all you have: watching the internet from 550 ms". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 137–150.

[Pra+16] J. Prados-Garzon, O. Adamuz-Hinojosa, P. Ameigeiras, J. J. Ramos-Munoz, P. Andres-Maldonado, and J. M. Lopez-Soler. "Handover implementation in a 5G SDN-based mobile network architecture". In: *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE. 2016, pp. 1–6.

[PTK94] S. Pingali, D. Towsley, and J. F. Kurose. "A comparison of sender-initiated and receiver-initiated reliable multicast protocols". In: *ACM SIGMETRICS Performance Evaluation Review* 22.1 (1994), pp. 221–230.

[RH19] K. Rose and J. Holland. *Asymmetric Loss-Tolerant Authentication*. Internet-Draft draft-krose-mboned-alta-01. Work in Progress. Internet Engineering Task Force, July 2019. 10 pp.

[RHB18] A. Rabitsch, P. Hurtig, and A. Brunstrom. "A stream-aware multipath QUIC scheduler for heterogeneous paths". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 2018, pp. 29–35.

[Riz00] L. Rizzo. "pgmcc: a TCP-friendly single-rate multicast congestion control scheme". In: *ACM SIGCOMM Computer Communication Review* 30.4 (2000), pp. 17–28.

[RJ88] K. Ramakrishnan and R. Jain. "A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer". In: *ACM SIGCOMM Computer Communication Review* 18.4 (1988), pp. 303–313.

[RKT98] D. Rubenstein, J. Kurose, and D. Towsley. "Real-time reliable multicast using proactive forward error correction". In: *Proceedings of NOSSDAV'98*. 1998, pp. 279–293.

[Rob88] L. Roberts. "The Arpanet and computer networks". In: *A history of personal workstations*. 1988, pp. 141–172.

[Roc+17] V. Roca, B. Teibi, C. Burdinat, T. Tran, and C. Thienot. "Less latency and better protection with AL-FEC sliding window codes: A robust multimedia CBR broadcast case study". In: *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE. 2017, pp. 1–8.

[RS60] I. S. Reed and G. Solomon. "Polynomial codes over certain finite fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.

[RSM18] V. Roca, I. Swett, and M.-J. Montpetit. "Sliding window random linear code (RLC) forward erasure correction (FEC) schemes for QUIC". In: (2018).

[RV98] L. Rizzo and L. Vicisano. "RMDP: An FEC-based reliable multicast protocol for wireless environments". In: *ACM SIGMOBILE Mobile Computing and Communications Review* 2.2 (1998), pp. 23–31.

[Sac+18a] J. Sachs, L. A. Andersson, J. Araújo, C. Curescu, J. Lundsjö, G. Rune, E. Steinbach, and G. Wikström. "Adaptive 5G low-latency communication for tactile internet services". In: *Proceedings of the IEEE* 107.2 (2018), pp. 325–349.

[Sac+18b] J. Sachs, G. Wikstrom, T. Dudda, R. Baldemair, and K. Kittichokechai. "5G radio network design for ultra-reliable low-latency communication". In: *IEEE network* 32.2 (2018), pp. 24–31.

[Sav+99] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. "TCP congestion control with a misbehaving receiver". In: *ACM SIGCOMM Computer Communication Review* 29.5 (1999), pp. 71–78.

[SBB05] R. Sherwood, B. Bhattacharjee, and R. Braud. "Misbehaving TCP receivers can cause Internet-wide congestion collapse". In: *Proceedings of the 12th ACM conference on Computer and communications security*. 2005, pp. 383–392.

[Sch21] D. Schinazi. *The MASQUE Protocol*. Internet-Draft draft-schinazi-masque-protocol-03. Work in Progress. Internet Engineering Task Force, Mar. 2021. 14 pp.

[See25] M. Seemann. *QUIC Interop Runner*. Accessed on March 27, 2025. 2025.

[SH26] M. X. Syona Sarma JQ Lau and V. Hwang. "Inside Gen 13: how we built our most powerful server yet". 2026.

[Sho06] A. Shokrollahi. "Raptor codes". In: *IEEE transactions on information theory* 52.6 (2006), pp. 2551–2567.

[SI20] M. Seemann and J. Iyengar. "Automating QUIC interoperability testing". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 2020, pp. 8–13.

[SMR18] I. Swett, M.-J. Montpetit, and V. Roca. "Coding for QUIC". In: (2018).

[SR] *segment-routing.net*. URL: `https://www.segment-routing.net` (visited on 08/30/2024).

[ST16] A. Schaub and D. Trey. `https://reproducingnetworkresearch.wordpress.com/2016/05/30/cs-244-16-misbehaving-tcp-receivers-can-cause-internet-wide-congestion-collapse/comment-page-1/`. CS244 '16: Misbehaving TCP receivers can cause Internet-wide congestion collapse. May 2016.

[Sto+20] 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.

[Sun+11] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros. "Network coding meets TCP: Theory and implementation". In: *Proceedings of the IEEE* 99.3 (2011), pp. 490–512.

[SYJ19] M. A. Siddiqi, H. Yu, and J. Joung. "5G ultra-reliable low-latency communication implementation challenges and operational issues with IoT devices". In: *Electronics* 8.9 (2019), p. 981.

[Tia+24] H. Tian, C. Xie, E. Chingwena, S. Peng, and Q. Gao. *SRv6 Deployment Consideration*. Internet-Draft draft-tian-spring-srv6-deployment-consideration-08. Work in Progress. Internet Engineering Task Force, Feb. 2024. 30 pp.

[Tre+21] M. Trevisan, D. Giordano, I. Drago, and A. S. Khatouni. "Measuring HTTP/3: Adoption and performance". In: *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*. IEEE. 2021, pp. 1–8.

[Tyu+22] N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette. "A high-speed quic implementation". In: *Proceedings of the 3rd International CoNEXT Student Workshop*. 2022, pp. 20–22.

[TZ01] W.-T. Tan and A. Zakhor. "Video multicast using layered FEC and scalable compression". In: *IEEE Transactions on circuits and systems for video technology* 11.3 (2001), pp. 373–386.

[Uni25] Unigamesity. "How big is Fortnite?" 2025.

[Vig+10] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock. "Dr. multicast: Rx for data center communication scalability". In: *Proceedings of the 5th European conference on Computer systems*. 2010, pp. 349–362.

[Wan+04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[Wel22] M. Welzl. "Not a Trade-Off: On the Wi-Fi Energy Efficiency of Effective Internet Congestion Control". In: *2022 17th Wireless On-Demand Network Systems and Services Conference (WONS)*. IEEE. 2022, pp. 1–4.

[Wel82] E. Weldon. "An improved selective-repeat ARQ strategy". In: *IEEE Transactions on Communications* 30.3 (1982), pp. 480–486.

[WH01] J. Widmer and M. Handley. "Extending equation-based congestion control to multicast applications". In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. 2001, pp. 275–285.

[WHB08] D. Wischik, M. Handley, and M. Bagnulo Braun. "The resource pooling principle". In: *ACM SIGCOMM Computer Communication Review* 38.5 (Oct. 2008), pp. 47–52.

[Wie+03] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. "Overview of the H. 264/AVC video coding standard". In: *IEEE Transactions on circuits and systems for video technology* 13.7 (2003), pp. 560–576.

[Wir+20] T. Wirtgen, Q. De Coninck, R. Bush, L. Vanbever, and O. Bonaventure. "Xbgp: When you can't wait for the ietf and vendors". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020, pp. 1–7.

[XDB18] M. Xhonneux, F. Duchene, and O. Bonaventure. "Leveraging ebpf for programmable network functions with ipv6 segment routing". In: *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 2018, pp. 67–72.

[Xu+15] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. "Investigating transparent web proxies in cellular networks". In: *Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings 16*. Springer. 2015, pp. 262–276.

[Yua+24] G. Yuan, M. Sotoudeh, D. K. Zhang, M. Welzl, D. Mazières, and K. Winstein. "Sidekick:{In-Network} Assistance for Secure {End-to-End} Transport Protocols". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 1813–1830.

[Zen+21] G. Zeng, L. Chen, B. Yi, and K. Chen. "Optimizing Tail Latency in Commodity Datacenters using Forward Error Correction". In: *CoRR* abs/2110.15157 (2021). arXiv: 2110.15157.

[Zir+21] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle. "It's over 9000: analyzing early QUIC deployments with the standardization on the horizon". In: *Proceedings of the 21st ACM Internet Measurement Conference*. 2021, pp. 261–275.

[ZY21] Z. Zhou and X. Yang. "Speeding Up TCP with Selective Loss Prevention". In: *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE. 2021, pp. 1–6.

[ZZ11] L. Zhang and B. Zhang. "The Novel Frame Boundary Detection and Fast Frame Synchronous Structure for 10 Gb/s Ethernet Phy FEC Sub-Layer VLSI Implementation". In: *2011 7th International Conference on Wireless Communications, Networking and Mobile Computing*. IEEE. 2011, pp. 1–4.