

École polytechnique de Louvain

Evaluating Flexicast QUIC through Real-World applications

Author: **Corentin DETRY**
Supervisor: **Olivier BONAVENTURE**
Readers: **Louis NAVARRE, Tom BARBETTE**
Academic year 2025–2026
Master [120] in Computer Science

Abstract

Multicast is an efficient scheme for one-to-many communication, but widespread deployment on the Internet has long been slowed down by reliability and security, as well as pricing concerns leading to the *chicken-or-the-egg* problem. Flexicast QUIC (FCQUIC) addresses some of these issues by extending Multipath QUIC to allow a source to create a shared multicast flow along with per-client unicast paths, providing secure and reliable multicast to a modern transport protocol.

In this thesis, we evaluate Flexicast QUIC through two contributions. We first implemented a multicast chat application using FCQUIC, and evaluated it against baselines built with QUIC and TCP (with and without TLS). The applications are evaluated on virtual network topologies of varying complexity, measuring end-to-end latency, source CPU utilization, and downstream goodput. FCQUIC’s implementation overhead is measurable in small topologies, but is quickly absorbed by the multicast-induced latency gains as the number of receivers, message size, or network loss increases.

We then address the *acknowledgment implosion* problem in Flexicast QUIC. We propose a hierarchical *relay* architecture in which relay nodes act as intermediate sources, redistributing data on new multicast flows, aggregating acknowledgments from downstream clients, and handling retransmissions locally. Two designs are presented, and one is implemented in two versions. We evaluate the relay implementations on Grid’5000 across 5 clusters, connected using GRE tunnels to create a multicast distribution tree on top of a non-multicast network. This is the first large-scale evaluation of Flexicast QUIC on real, geographically distributed servers rather than fully emulated topologies or smaller-scale deployments.

Acknowledgments

I would like to express my most sincere gratitude to my promoter, Professor Olivier Bonaventure, and to Louis Navarre, for offering me the opportunity to work on such an interesting topic and for their invaluable insight and feedback throughout the year.

My thanks also go to Anthony Doeraene, whose help allowed me to progress on multiple occasions, and whose scripts for simulated topologies in network namespaces were used for several evaluations in this work.

I am also grateful to Paul-Edouard Bertrand Van Ouytsel for his feedback and to Donatien Schmitz for his explanations and insights into Grid'5000.

Finally, I would like to express my heartfelt thanks to my family and friends for their support throughout my studies and during this master's thesis.

Glossary

ASM – Any Source Multicast: Multicast model where any host can send to a multicast group

FCQUIC – Flexicast QUIC: Flexible multicast extension of QUIC and Multipath QUIC

GRE – Generic Routing Encapsulation: Tunneling protocol that encapsulates packets inside of IP packets

IGMP – Internet Group Management Protocol: IPv4 multicast group membership protocol

MoQ – Media over QUIC: Low latency media delivery solution for QUIC

NPF – Network Performance Framework: Framework for running reproducible network experiments

OSPF – Open Shortest Path First: Link state routing protocol

PIM – Protocol Independent Multicast: Family of multicast routing protocols

PIM-SM – PIM Sparse Mode: PIM variant that creates shared source trees using rendezvous points

PIM-SSM – PIM Source-Specific-Multicast: PIM variant that creates source trees rooted at the specific source

Quiche: Cloudflare's QUIC implementation

SSM – Source Specific Multicast: Multicast model where only a specific source can send to a group

TCP – Transmission Control Protocol: Reliable and connection oriented transport protocol

TLS – Transport Layer Security: Cryptographic protocol that secures transport protocol connections

UDP – User Datagram Protocol: Unreliable and connectionless transport protocol

Contents

1	Context and Motivations	1
2	State of the Art	3
2.1	Multicast	3
2.1.1	Multicast Tunneling	4
2.2	QUIC	6
2.3	Multipath QUIC	7
2.4	Flexicast QUIC	7
2.4.1	Architecture	8
2.4.2	Performance	9
2.4.3	tokio-fcquiche	10
3	A Multicast chat	11
3.1	Multicast for Content Distribution	11
3.2	How does a multicast chat application work?	12
3.2.1	Messages	13
3.3	Flexicast QUIC application	14
3.3.1	Application architecture	14
3.3.2	Rust Implementation	15
3.4	Baseline QUIC application	15
3.4.1	Application architecture	15
3.4.2	Rust implementation	16
3.5	TCP application	17
3.5.1	Application architecture	17
3.5.2	Performance tuning	18
3.5.3	Rust Implementation	19
3.6	Evaluation	19
3.6.1	Testing environment	19
3.6.2	Testing methodology	20
3.6.3	Topologies	20
3.6.3.1	Solo: One client, one router, no delay	21
3.6.3.2	Tiny: Two clients, one router, no delay	21
3.6.3.3	Small: Three clients, three routers, no delay	21
3.6.3.4	Medium: 28 clients, seven routers, 1ms delay per link	22
3.6.3.5	Receivers and Data Test Topologies	23
3.7	Results	23
3.7.1	Network condition tests	23
3.7.2	Receivers Scalability	30
3.7.3	Payload sensitivity	32
3.8	Conclusion	35
4	Flexicast QUIC Relay	38
4.1	Motivation: Acknowledgment implosion	38

4.1.1	Proposed Solution: Hierarchical Relay Architecture	39
4.2	Implicit Relay Design	40
4.2.1	Message Caching	40
4.2.2	Benefits and Trade-offs	41
4.2.3	Modifications to Flexicast QUIC	41
4.3	Implicit Relay implementation	42
4.3.1	Application-level Relay	42
4.3.1.1	Architecture	42
4.3.1.2	Rust Implementation	42
4.3.2	tokio-fcquiche Relay	44
4.3.2.1	Architecture	44
4.3.2.2	Rust Implementation	44
4.4	Conclusion of relay design and implementation	45
4.5	Multi-site evaluation on Grid'5000	46
4.5.1	Inter-site connectivity	46
4.5.2	Multicast support on Grid'5000	46
4.5.3	Experimental setup	47
4.5.4	Relay Evaluation	48
4.5.4.1	Tests and configurations	48
4.5.4.2	Topology	49
4.5.5	Results	50
4.5.5.1	Relay overhead	50
4.5.5.2	Data segmentation	52
4.6	Conclusion	54
5	Future evolution: Explicit Relay Design	56
5.1	Design	56
5.2	Congestion Control with relays	57
5.3	Relay migration	57
5.4	Benefits and Trade-offs	58
5.5	Conclusion	58
6	Conclusion	59
6.1	Multicast Chat	59
6.2	Flexicast QUIC Relays	60
6.3	Source code	61
	Generative AI Disclosure	62
	Bibliography	63
A	Appendix	68
A.1	Modifications to Flexicast QUIC for the Implicit Relay	68
A.2	Modifications to Flexicast QUIC for the Explicit Relay	68
A.3	Topology script configuration	70
A.3.0.1	Example topology file	72

Chapter 1

Context and Motivations

Many widely used internet applications are one-to-many: live-streaming, software and video game downloads, video conferencing, live news articles, and many other use cases all come down to a source distributing the same content to many receivers [1]. To distribute this traffic, many applications rely on unicast routing protocols, forcing them to send as many as N copies of the data for N receivers.

IP Multicast, which applies the multicast scheme to Internet Protocol networks [2], was proposed in the early 1990s to provide efficient distribution to internet applications. Despite the clear use cases, however, adoption and deployment in large networks have been slow due to a combination of pricing difficulties, lack of reliability mechanisms, and security concerns [3], [4]. As a result, IP multicast has typically only been deployed for intra-domain use cases, such as IPTV [5], software updates in enterprise networks [6], and video conferencing. Inter-domain multicast deployments have remained rare, in part due to the lack of popular applications requiring ISPs to deploy multicast; such applications, in turn, never developed due to the lack of native multicast support, resulting in a typical *chicken or the egg* problem.

The recent growth in live-video streaming has caused issues even for large media streaming companies, as illustrated by Netflix’s poor live stream quality during live sporting events [7]. Research into deployable, reliable, and safe multicast solutions has been driven, in part, by such issues. Protocols such as Automatic Multicast Tunneling (AMT) [8] and DRIAD [9] have been standardized to enable incremental inter-domain multicast deployment, and researchers are now actively bringing multicast to transport protocols, with efforts such as a multicast extension for QUIC [10] and Flexicast QUIC, an extension of Multipath QUIC that creates a shared multicast flow as a unidirectional additional path [11].

Evaluations of Flexicast QUIC (FCQUIC) have demonstrated its scalability and robustness for file delivery and live streams to many concurrent receivers [11]. FCQUIC has also been tested in an inter-domain scenario, using AMT to encapsulate traffic from the source’s multicast network to other multicast islands [4]. A few questions remain, however: first, whether FCQUIC is a suitable solution for one-to-many or many-to-many applications that do not transmit large amounts of data (e.g., stock market updates or distributed database replication), and second, whether FCQUIC scales as well for such applications as it does for live streams and file delivery.

In this master’s thesis, we address these questions through two contributions. First, we build a multicast chat application using Flexicast QUIC, along with baselines using QUIC and TCP, and evaluate them across different scenarios to get a complete picture of Flexicast QUIC’s advantages and disadvantages for such applications. We then increase the scale of the application and evaluate how it scales with a large number of receivers in such an application. Second, to mitigate the *acknowledgment implosion* problem that plagues reliable multicast protocols, we design a hierarchical relay structure, an architectural change in which relay nodes act as intermediary sources between the FCQUIC source and the downstream clients. Two implementations of this design are evaluated: one built using the primitives exposed to an FCQUIC user, and the other integrated directly into FCQUIC. The comparison aims to evaluate whether an end user of a reliable multicast protocol like FCQUIC can address this issue themselves, or whether they require more complex solutions built into the protocol.

Chapter 2

State of the Art

In this first chapter, we introduce some important protocols and concepts used and built upon during this work. First, we explain IP Multicast, its deployment issues, and some deployment techniques that will be used in this work. Second, we describe QUIC and some of its important differences to well-known transports like TCP. Third, we go over the QUIC extension Multipath QUIC. And fourth, we describe Flexicast QUIC, a multipath QUIC extension that brings flexible multicast to QUIC and which is the focus of this work.

2.1 Multicast

Multicast is a one-to-many or many-to-many communication scheme, where transmissions are sent to a *group address* representing a collection of receivers [12]. Multicast distribution scales to large numbers of receivers thanks to efficient packet replication, ensuring that each data packet of a multicast tree is sent once per network link [13].

IP Multicast, first proposed by Deering and Cheriton [2], applies multicast to Internet Protocol (IP) networks. Multicast group addresses are located in the IPv4 224.0.0.0/4 range and hosts express their interest in leaving or joining groups using a group membership protocol such as the Internet Group Management Protocol (IGMP) [14].

The source and the set of receivers for a multicast group address form a *multicast distribution tree* in the network, along which multicast data packets flow. Several types of trees can be built, differing in where their roots lie and in how widely they span the network. Upon reception of an IGMP join request, routers need to join the desired *multicast tree*. This can be achieved using the Protocol Independent Multicast (PIM) protocol, among others. As its name implies, PIM relies on existing unicast routing protocols to discover and react to changes in the network topology [12]. PIM exists in four variants, each responsible for the creation of a specific type of multicast tree under different assumptions about receiver location and knowledge of the source(s). Common to all variants is the way the state is maintained on the routers. A multicast tree is identified by a pair (S, G) , where S is the source address, and G the group address. Additional information, such as the router's incoming interface (IIF) for the multicast traffic and the outgoing interface set (OIF), is also maintained. In this schema, the source address S may be omitted from the receivers' request: $(*, G)$, allowing receivers to

express interest in transmission from any source to the group address G , this is known as Any-Source-Multicast [13].

Source-Specific-Multicast (SSM, RFC4607 [13]) restricts this scheme to one-to-many communications by only allowing a transmission sent with source address S and destination address G to be forwarded and delivered to hosts having explicitly requested to join the tree (S, G) [13]. Each multicast tree is uniquely identifiable by both the source address S and group address G , this allows multiple applications to use the same SSM source address S concurrently in the network without conflicts [13]. The set of all IIF and OIF for a pair (S, G) in all the routers form a reverse Shortest Path Tree (SPT) rooted at the source S [12]. PIM-SSM, a variant of PIM used for SSM, creates source trees rooted at a specific source S , disallowing any other host from sending on this multicast tree [15].

IP Multicast has been deployed for use cases such as IPTV and stock exchange information distribution. Its use on the Internet, however, has been nearly nonexistent. Although live streaming, video conferencing, and software distribution are use cases that may benefit from multicast distribution, deployment and pricing difficulties, as well as security and reliability have dissuaded Internet Service Providers and cloud providers from adopting the service model [3], [4].

2.1.1 Multicast Tunneling

Multicast enabled networks may often be only interconnected with non-multicast networks, forming stranded *multicast islands* [4]. Multicast traffic cannot flow between these islands natively. However, using protocols such as Automatic Multicast Tunneling (AMT) or Generic Routing Encapsulation (GRE) to encapsulate multicast traffic inside unicast traffic [16], we can establish tunnels so that hosts in these islands can use multicast to communicate with each other.

Generic Routing Encapsulation. Generic Routing Encapsulation (GRE, RFC 2784 [17]) is a tunneling protocol that can encapsulate packets from any protocol inside another protocol. GRE must be used between two manually configured endpoints, forming a point-to-point tunnel. Endpoints must be able to reach each other. The endpoint configuration is static (i.e. endpoints cannot discover each other dynamically) and must be performed before any traffic can be sent through the tunnel [16]. Once encapsulated, a GRE packet appears as a regular unicast packet traveling from one end of the tunnel to the other. From the perspective of the tunnel endpoints, however, the tunnel behaves as a typical link, allowing routers to run unicast or multicast routing protocols over it. The GRE header contains a 16 bit protocol type field that identifies the encapsulated packet's protocol. GRE's overhead is typically 24 bytes (including the delivery IPv4 header) when no

checksum is included, in which case the GRE header itself is only 4 bytes long. See Figure 1 for the structure.

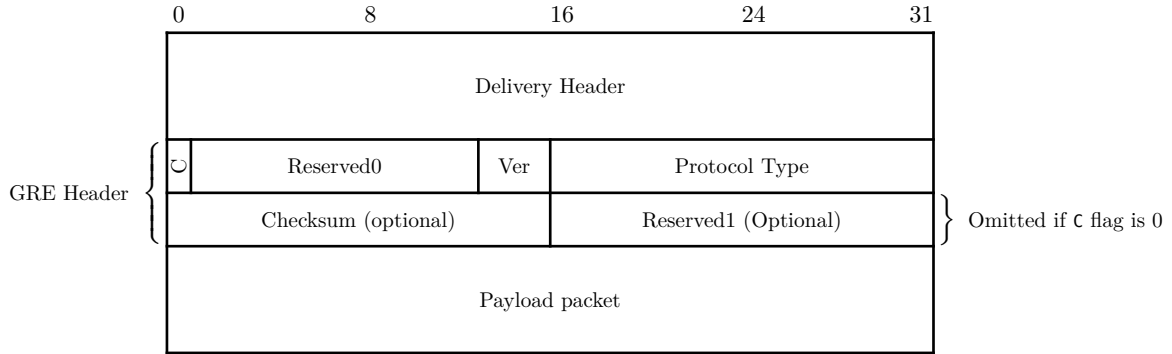


Figure 1: Structure of a GRE encapsulated packet [17]

Figure 2 illustrates a GRE tunnel between routers R_1 and R_2 , each located in a different multicast-enabled network. Clients C_1 and C_2 in the isolated site (on the right) use this tunnel to receive multicast content from a source S located in another network (on the left), even though the network connecting the two sites does not support multicast.

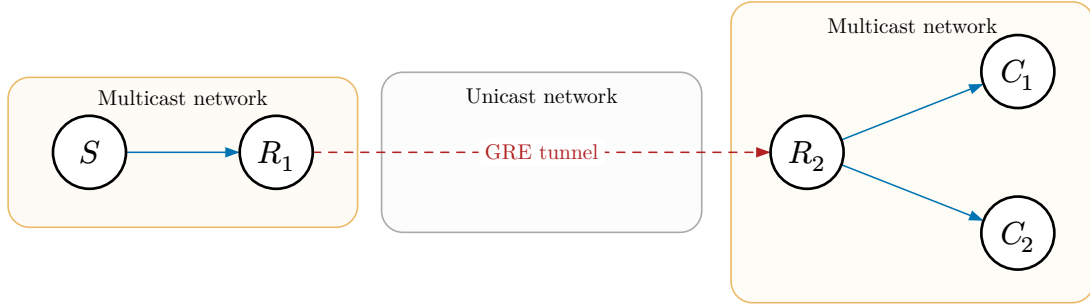


Figure 2: GRE tunnel between two multicast-enabled networks (inspired by AMT RFC [8])

Automatic Multicast Tunneling. Automatic Multicast Tunneling (AMT) defined in RFC 7450 [8], provides automatic multicast traffic encapsulation, allowing this traffic to originate from, or be routed through, networks that don't support multicast. AMT introduces two roles to achieve this: the AMT gateway, which is placed near the multicast receivers, and the AMT relay, which is placed in a multicast source's network. The gateway may be running on the host itself when located in a non-multicast site. The relay receives requests from gateways for traffic available in its network, joins the requested multicast tree, and forwards the encapsulated packets to the gateway [8]. AMT is more flexible than GRE, as it does not require manual configuration, however, it does add more software or devices in the multicast forwarding path [16].

AMT is a core component of TreeDN [18], a tree-based Content Delivery Network (CDN) architecture that lets network operators live stream to massive audiences at little additional cost. Within the TreeDN architecture, AMT allows receivers in unicast-only networks to receive multicast content sent by an AMT relay located at the boundary of the operator’s network [18].

2.2 QUIC

QUIC [19] is a modern, connection-oriented transport protocol that runs atop UDP and provides reliable, secure, and multiplexed delivery of messages. Although QUIC may appear similar to TCP due to its reliability and connection-oriented nature, it differs in that it provides built-in TLS encryption, head-of-line-blocking avoidance thanks to stream multiplexing, and faster handshakes [20].

QUIC packets are authenticated and encrypted using TLS 1.3, with a handshake designed to minimize latency [20]. QUIC cleartext packet headers are short, containing only a few flags and a connection ID. Everything else, such as Packet Numbers (PN), is encrypted. To allow for more flexibility, QUIC did away with TCP’s 4-tuple as a means of identifying a connection, relying instead on *connection IDs* chosen by both end hosts [20]. Each end host keeps track of distinct sequence number spaces [21] and every transmitted packet is assigned its own unique PN. Retransmissions of lost packets use new packet numbers. Besides the cleartext header, everything is encrypted, thereby reducing the possible impact of middleboxes that do not implement the latest protocol version on the protocol evolution [20].

Inside the encrypted contents of the packet is a series of *frames*, which may carry control information (e.g., `ACK`, `MAX_DATA`, `PATH_CHALLENGE`,... frames), or data (e.g., `STREAM`, `DATAGRAM` frames). When a QUIC packet is lost, a new packet (with a new packet number) containing the lost frames is transmitted [19].

Streams. A stream is an abstraction representing a unidirectional or bidirectional communication channel within a QUIC connection [19]. Many different streams may be transmitted over a QUIC connection at any given moment. Each stream is uniquely identified by a *Stream ID*, a 62-bit integer, allowing over 4 quintillion ($2^{62} - 1$) unique streams to be identified [19]. Stream contents are carried by `STREAM` frames. A single frame may open, send data for, and close a short-lived stream, or may carry only part of a long-lived stream. Head-of-line blocking avoidance in QUIC is provided thanks to each stream being independent. The loss of a stream’s frames will only delay this stream from progressing, while other streams may continue as usual [19]. Therefore, a web page that loads each of its resources using a separate stream

(e.g., one file per stream) will finish downloading faster over QUIC than over TCP in lossy scenarios.

2.3 Multipath QUIC

Multipath QUIC [22] is a QUIC extension that enables a connection to use multiple network paths (e.g., one path over wifi and another over LTE) [23]. Each path has its own packet number space (i.e., packet numbers are only meaningful for a specific path) and is uniquely identified by a monotonically increasing *Path ID*, starting at 0. Each connection ID is assigned to exactly one path ID, but a path may have multiple connection IDs. For an end host, the connection ID identifies which path the packet belongs to, and will put in context the packet number [22]. Using separate packet number spaces per path enables multipath QUIC to use QUIC’s loss detection and congestion control mechanisms per path [24]. However, data or control information is not tied to a particular path, allowing a host to send an acknowledgment on a path for data received on another [11].

Two Multipath QUIC-supporting endhosts negotiate usage of multiple paths through the inclusion of the `initial_max_path_id` transport parameter. This parameter is used as an indication of multipath QUIC support, but also includes the initial maximum number of paths that may be used when communicating with the host/endpoint [22]. This limit may be raised later during the connection by sending a `MAX_PATH_ID` frame with the new maximum path ID supported by the host, thereby allowing more paths to be created.

A client may create a new path through QUIC’s `PATH_CHALLENGE` peer validation process. New paths may not be accepted by the other remote host, in which case they may send a `PATH_ABANDON` frame. `PATH_ACK` frames are introduced by multipath QUIC to allow hosts to acknowledge data for a path *i* over another path *j* [22], [25].

Given that each path maintains its own packet number space, they must also maintain their own congestion control state and loss recovery information. Using traditional congestion control schemes may cause issues if the same underlying network link is used for multiple paths, thereby requiring multipath specific congestion schemes to avoid unfairness [23].

2.4 Flexicast QUIC

Flexicast QUIC (FCQUIC) [11] is a proposed QUIC and Multipath QUIC extension that brings *flexible multicast* to a modern transport layer protocol. An FCQUIC connection combines per-receiver bidirectional unicast paths with shared unidirectional *multicast flows*. An FCQUIC source uses multicast

flows as its primary data transfer channel, with client acknowledgments and retransmissions occurring on the bidirectional unicast paths [11]. Support for FCQUIC is negotiated during the handshake through the `flexicast_support` transport parameter as well as multipath QUIC’s `initial_max_path_id` parameter [25].

2.4.1 Architecture

Multicast flow and unicast paths. A multicast flow is a unidirectional path whose destination address may be a multicast group address. Each multicast flow is identified by a *Flow ID*, equivalent to a Connection ID [25]. When multicast is not supported by the underlying network, the source may replicate the encrypted packet contents and transmit them to each receiver using `sendmsg`. Unicast paths are established by receivers through a standard QUIC handshake and maintained for the lifetime of the connection, allowing receivers to acknowledge multicast flow packets and receive retransmitted packets [25]. The multicast flow uses a separate set of keys derived by the source alone and sent to receivers over their (secure) unicast paths.

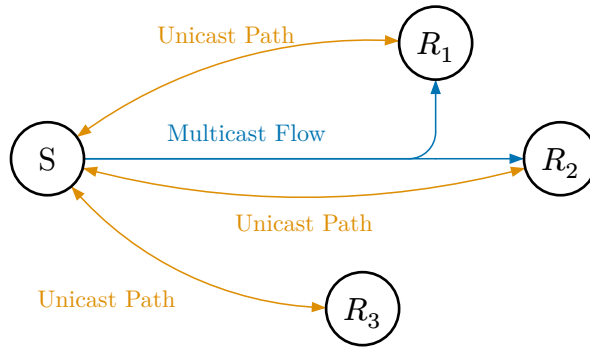


Figure 3: Flexicast QUIC paths overview (inspired by [11])

Figure 3 presents this architecture. Receivers R_1 , R_2 , and R_3 opened unicast paths to the source S , negotiated Flexicast QUIC support, received the required information and keys about the multicast flow, but only R_1 and R_2 joined the multicast flow. Receiver R_3 illustrates a receiver located in a non-multicast network that relies on the unicast path for data delivery. This unicast *fallback* mode does not scale as well as multicast delivery [11], [25].

The unicast fallback mechanisms may also be used by the source when a receiver is acknowledging multicast flow data too slowly. Given the shared nature of the multicast flow, its throughput must be one that all receivers can sustain, i.e., the sending bitrate must not be higher than the receiving bitrate of the slowest receiver. To prevent a single slow receiver from lowering the multicast flow’s throughput to an unacceptable rate, the source may force

this receiver to fall back to its unicast path, allowing the remaining, faster, receivers to benefit from a higher sending rate on the multicast flow [11].

Flow advertisement. The multicast flow is advertised to receivers with the `FC_ANNOUNCE` frame, which contains the flow ID, the source and multicast addresses, as well as the UDP destination port and the acknowledgment delay [11]. To join the multicast flow, a receiver signals its intent to the source with an `FC_STATE(JOIN)` frame. This will prompt the source to send the shared key used for the multicast flow in an `FC_KEY` frame. Once this information is received, the receiver will join the multicast tree using IGMP or MLD. However, given the unidirectional nature of the multicast flow, the receiver cannot perform peer validation through the `PATH_CHALLENGE` process, instead, because it received this information over a secure unicast path, it trusts the information and does not perform peer validation. Once the receiver has joined the multicast tree, it informs the source with an `FC_STATE(LISTENING)` frame. If, however, the receiver was not able to join the multicast tree, it will never send the `FC_STATE(LISTENING)` frame and the source, after a timeout, will tell it to fallback to unicast, or to use another multicast flow that uses replication instead [11], [26].

A receiver may leave a multicast flow at any moment by sending the `FC_STATE(LEAVE)` frame, however, this does not stop the source from transmitting data to this receiver over its unicast path. It may, for example, decide to join another multicast flow with a different throughput [26].

Reliability and acknowledgments. Since FCQUIC extends Multipath QUIC, receivers acknowledge data with `PATH_ACK` frames. These frames carry the path identifier on which the ACKed data was transmitted. In FCQUIC, the multicast flow is unidirectional, therefore, the receivers acknowledge multicast flow data with `PATH_ACK` frames on the unicast path [25].

To limit this acknowledgment rate, the source can include an *acknowledgment delay* in the `FC_ANNOUNCE` frame. This value is used by the receivers to randomize the interval at which they acknowledge data to the source, to try to reduce the overall acknowledgment rate due to the value acting as a minimum interval [11], [25].

2.4.2 Performance

FCQUIC evaluations performed on CloudLab [27] showed that, with an 80 Mbps outgoing stream, an FCQUIC source could sustain over 1000 receivers and deliver more than 78 Gbps of traffic, while keeping source CPU usage between 50% and 75% [11]. In the same experiment, a unicast QUIC baseline only sustained 200 receivers, delivering 20 Gbps of traffic [11].

2.4.3 `tokio-fcquiche`

The Flexicast QUIC implementation is built on top of a fork of Cloudflare’s `quiche` [28] that supports Multipath QUIC [29]. A crate of this implementation, `tokio-fcquiche`, allows easy development of Flexicast QUIC applications. It abstracts away most protocol implementation details and exposes `tokio` channels [30] that allows the application to send or receive data from the multicast flow.

Internally, the library is split into many `tokio` tasks, each communicating through message passing via channels [11], [29]. An I/O task receives incoming packets and forwards them to the right unicast path handler. A multicast flow handler task receives messages to send on the multicast flow, encrypts them using the shared multicast flow keys, then transmits them on the socket [11].

Chapter 3

A Multicast chat

This section presents the implementation and evaluation of a multicast chat application. The applications are built on top of Flexicast QUIC’s `tokio-fcquiche` library [29], Cloudflare’s `quiche` library [28], and TCP. We first describe the different application implementations and their differences, then introduce the evaluation testbed and virtual topologies. We finally discuss the performance results.

3.1 Multicast for Content Distribution

Many applications, such as live video streaming, software or video game distribution, distributed database systems, stock market updates [1] can all take advantage of multicast distribution for their one-to-many communication. However, despite clear use cases, due in part to a lack of multicast support in Internet transport protocols, many applications opted to rely on unicast transmission. Lack of reliability, security, and deployment difficulties have all slowed down multicast’s adoption [3]. Flexicast QUIC solves the reliability and security problems by using Multipath QUIC [23] to create a multicast flow while maintaining per-client unicast path for acknowledgments and retransmissions. Each unicast path is protected using a unique pair of keys, while the multicast flow is protected with a common key, sent over the (encrypted) unicast paths to the clients.

Conferencing and group communications are examples of applications that adapt easily to Source-Specific-Multicast (SSM) [13]. A known server creates a multicast tree. Participants use IGMP or MLD to join this tree identified by the server address S and multicast destination address G (group address). To communicate, participants can send audio and video data to the server using a unicast transport protocol. The server may distribute the information over the multicast tree if the participant is authorized. A multicast chat is a similar, but simpler scheme, with participants joining the multicast tree (S, G) , transmitting messages to the server via a unicast path, and receiving messages over the multicast tree. Figure 4 illustrates this scheme: client C sends a message to the server S over its unicast path, and S forwards it to every participant over the multicast tree (S, G) .

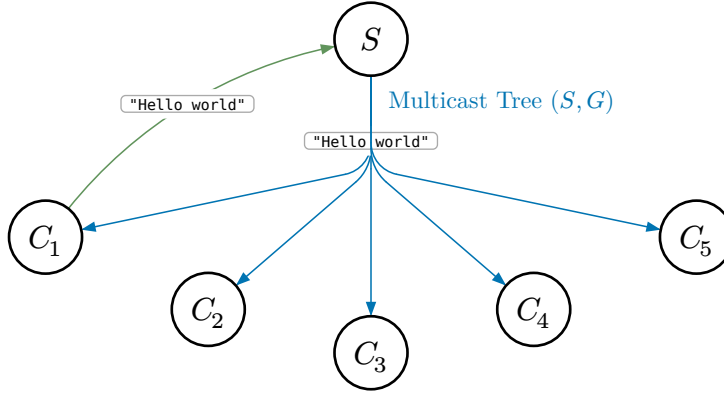


Figure 4: Multicast chat scheme illustration

3.2 How does a multicast chat application work?

To evaluate the performance improvements Flexicast QUIC may bring in terms of latency and source server utilization, as well as its potential drawbacks, a simple multicast chat scheme was defined:

1. Clients all connect to a single, shared server. Each client is identified by a unique ID (monotonically increasing number).
2. The server maintains state for each client and is able to (re)transmit information to each separately.
3. Only one client among them acts as the sender, typically Client 1. The sender sends messages according to a Poisson distribution to better approximate human generated network traffic. Each message contains the client username or ID, the current timestamp, and an optional random payload. Whenever the payload is empty, the message contains only the client ID and timestamp.

Measure of time. The evaluations are performed in network namespaces. All timestamps are obtained from a monotonic (non-decreasing) clock. This guarantees that any later reading is greater than or equal to a previous one. In the implementation this is provided by `std::time::Instant` [31].

Inter-packet interval. Network user “session” arrivals (i.e., network packet arrivals resulting from user actions) are often described using Poisson distribution [32]. This has been confirmed empirically [33] for user session arrivals, but is not a good model for other, slightly finer-scale arrivals [34]. It is, however, a good model for our evaluation, as a constant inter-packet interval would fail to be representative of real user behavior, but may also fail to highlight differences in queuing.

The inter-packet interval can either be defined as a constant interval (e.g., one packet every 100ms), or based on a Poisson distribution. A Poisson distribution with rate λ produces k events in a time window of length T with probability:

$$P(k \text{ events in interval } T) = \frac{(\lambda T)^k e^{-\lambda T}}{k!}, \quad \forall k \in \mathbb{N}$$

The inter-arrival times of events in the interval T are independent of each other, and are distributed according to an exponential distribution with mean $\frac{1}{\lambda}$.

Our sender therefore draws a value from an exponential distribution to obtain the inter-packet interval. The parameter λ , expressed in packets per second, dictates the packet rate, so $\lambda = 10^4$ corresponds to ten thousand packets per second on average.

Based on the packet size P and λ , one can estimate the minimum bandwidth required for a transmission. With packets of size $P = 1200$ bytes and $\lambda = 10^4$, the throughput is $(P \cdot 8) \cdot \lambda = 9.6 \cdot 10^7$ (ninety-six million bits per second, i.e., 96 Mbits per second)

3.2.1 Messages

Message payload. Messages can carry a fixed size random payload. The payload size is determined by a program flag. It is used to increase each message's size to be more representative of typical message sizes. The bytes are pseudo-random and allocated once by the sender. They induce no additional load during transmission on the client application.

Message format. Messages are encoded as a little-endian byte sequence. The implementation uses `rkyv` [35], a zero-copy (de)serialization framework for Rust. Figure 5 shows the message structure.

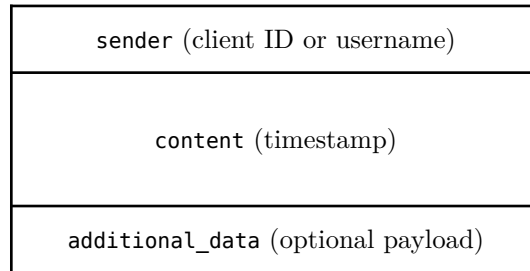


Figure 5: Contents of a chat message.

3.3 Flexicast QUIC application

This section presents the Flexicast QUIC multicast chat application. We first describe the application architecture on the client and the server, then discuss the implementation details related to the `tokio-fcquiche` library.

3.3.1 Application architecture

On the client side, as shown in Figure 6, the FCQUIC implementation exposes one channel for receiving messages from the server (either on the multicast flow or on the unicast path) and one for transmitting on the unicast path.

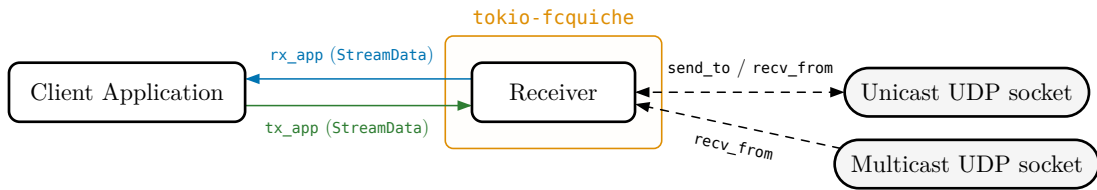


Figure 6: Flexicast QUIC client application architecture

The client application asynchronously waits for messages on the inbound channel. Upon reception, every receiver computes the total end-to-end latency based on the timestamp found in the message. This result is logged and captured by NPF [36]. The sender client transmits each message in a separate QUIC stream. These are client-initiated, unidirectional streams so that the sender isn't required to maintain receiving state.

On the server side, as shown in Figure 7, the FCQUIC implementation exposes one channel for receiving client messages on their unicast paths, and one for transmitting messages on the multicast flow. All incoming unicast messages share the same channel to the application, therefore, the client IDs in the messages are used by the application to identify users.

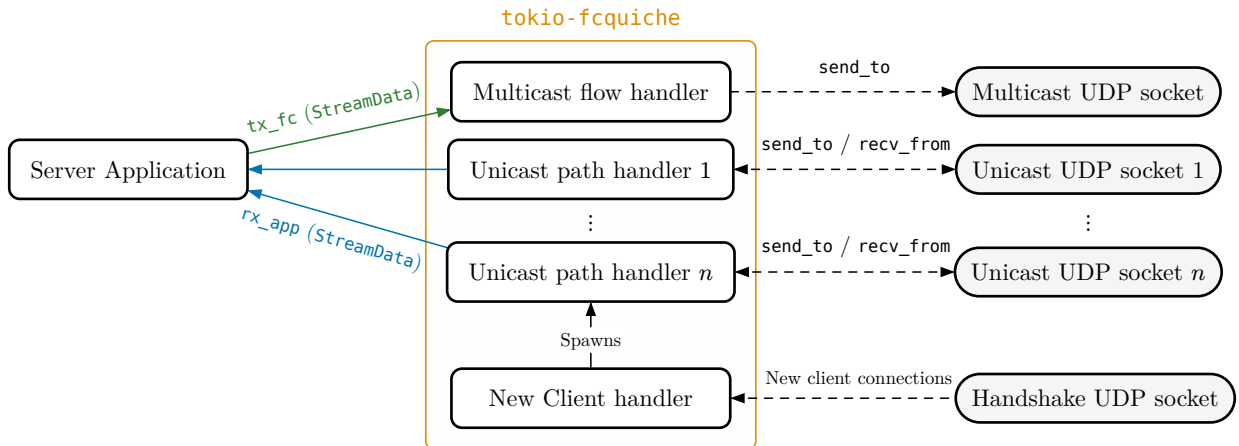


Figure 7: Flexicast QUIC server application architecture (inspiration: [11])

Upon reception of a message on the unicast channel from a client, the server creates a new message with the incoming message data and transmits it on the multicast flow channel. The multicast flow only carries messages in server-initiated, unidirectional streams so that the server does not have to maintain receiving state, given that no response can arrive on this path.

3.3.2 Rust Implementation

The application is built using the `tokio-fcquiche` library [29], which we extended to allow clients to send messages to the server on their unicast paths, as well as for the server to handle these incoming messages. The channels described above are `tokio` channels. On the server, the application asynchronously waits on both concurrently with an unbiased `tokio::select!` statement.

The `quiche` configuration is identical between client and server. All use the CUBIC congestion control algorithm, with pacing disabled, no Forward Error Correction (FEC), and no flow control limit (set to 100 GB). The acknowledgments delay (`max_ack_delay`) was left at the default 25ms.

Fall-back delay tuning. Flexicast QUIC provides a fall-back mechanism that marks clients that have not acked multicast packets during a certain time period (the fall-back delay D_{fb}). Once a client is marked as having fallen back, the server informs the client via the unicast path that all further transmissions will take place over this unicast path.

The FCQUIC fork maintained for this work (which diverged from the source at the end of December 2025) suffered from an intermittent issue that caused it to think that a multicast packet was never acknowledged early on during the connection. Investigations revealed this packet to be left over from the handshake, however, our debugging attempts were unsuccessful. To mitigate this issue, the fall-back delay D_{fb} was set to 1000 milliseconds, substantially higher than the round-trip latency in the test topologies.

3.4 Baseline QUIC application

To meaningfully derive conclusions from the evaluation of Flexicast QUIC, we compare it to a baseline QUIC application. The FCQUIC implementation is an extension of Cloudflare’s QUIC implementation called `quiche` [28], therefore, the baseline is also built with `quiche`.

3.4.1 Application architecture

The baseline client is similar to the FCQUIC client, it has access to one channel that receives messages from the server and one that it can use to transmit messages to the server. Figure 8 presents the client architecture.

The server uses a multitask architecture with a main task, per-client connection handlers, and a central controller that broadcasts incoming client messages to all other handlers. A single UDP socket is shared across the server: the main task accepts incoming connections and dispatches packets to the handler responsible for the connection, which forwards stream data to the controller task to be broadcast. Figure 9 presents this architecture.

3.4.2 Rust implementation

`quiche` provides two ways of developing applications: 1) using it directly and working with the underlying UDP socket, maintaining state, and handling multi-threading. Or 2) using `tokio-quiche` [37], which is a wrapper library around `quiche` that uses `tokio` as the asynchronous runtime. `tokio-quiche` manages the underlying UDP socket and handles state, thereby lowering the barrier of entry to QUIC applications [38].

We first built the baseline directly on `quiche` and `tokio` [30], heavily inspired by files from an older Flexicast QUIC version ([39], [40], [41]). This version was dropped because it had poor performance compared to the FC-QUIC application, even in situations where multicast provided no advantage and where Flexicast QUIC’s implementation complexity should have given it a comparatively worse performance. Several performance optimizations were applied to little avail. Given that these issues were likely caused by design or programming errors, we rebuilt the baseline using `tokio-quiche`, which connects `quiche`’s connection primitives to `tokio`’s event loop and abstracts away the inter-task coordination.

Applications built on `tokio-quiche` must create *drivers* by implementing the `ApplicationOverQuic` trait. The library provides a driver for HTTP/3 applications. For our multicast chat, we implement two drivers: a `ClientDriver` and a `ServerDriver`. These drivers store incoming and outgoing messages that have not yet been processed by the application. The methods implemented in the drivers are used as callbacks by the library, allowing us to perform certain actions at various parts of the connection’s lifecycle.

On the client, the application creates a UDP socket and hands it to `tokio-quiche` to obtain a `QuicheConnection` object. It is through this connection object that the `ClientDriver` communicates with the socket. The application communicates with the driver using two `tokio` channels, in an identical manner to the FCQUIC client.

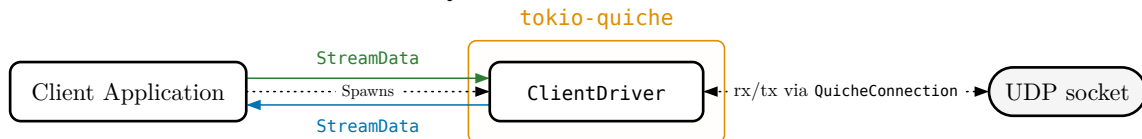


Figure 8: Baseline QUIC client application architecture

On the server, each client connection is handled by its own `ServerDriver`. The driver interacts with the client through a `QuicheConnection` object, in much the same way as the `ClientDriver`. The shared UDP socket is handed to `tokio-quiche` to create a connection listener. The main task accepts incoming connections, creates a new `ServerDriver`, and binds it to the newly created `QuicheConnection`. The channels required by the driver are then created and sent to the controller task, which broadcasts incoming client messages to all other `ServerDrivers`.

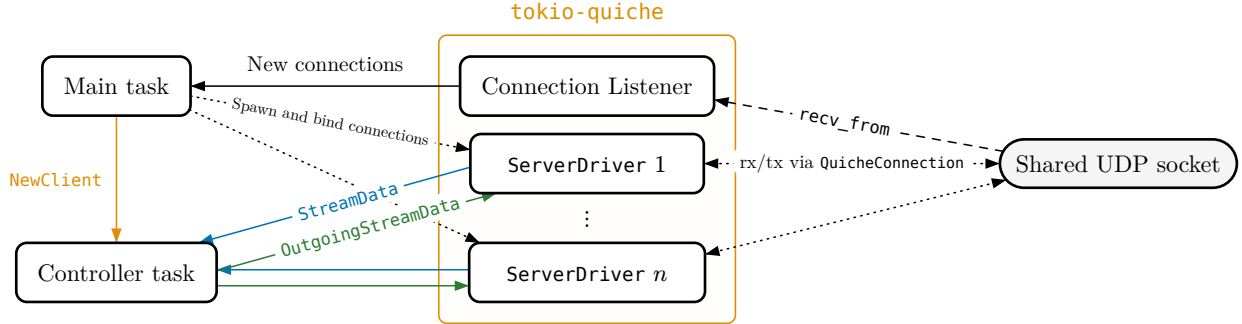


Figure 9: Baseline QUIC server application architecture

While these drivers maintain buffers, inter-task coordination is abstracted away by the library, which reduces the risk of introducing performance bugs.

3.5 TCP application

While QUIC provides many advantages over other transport protocols, such as shorter handshakes, elimination of head-of-line blocking, reduced impact of middleboxes on protocol evolution [20], TCP remains the dominant transport protocol [42]. Comparing our Flexicast QUIC and QUIC applications to a TCP baseline allows us to put latency figures for the QUIC-based implementation into perspective, and see, for example, when QUIC’s head-of-line blocking avoidance helped reduce the application’s latency.

3.5.1 Application architecture

The TCP server architecture is very similar to the previously presented architectures. A main task accepts incoming connections, optionally upgrades the stream to a TLS stream, then spawns a client handler task which reads and writes to the connection stream. Communication between client handler tasks is handled by a controller task, just like in Figure 9. Figure 10 presents this architecture.

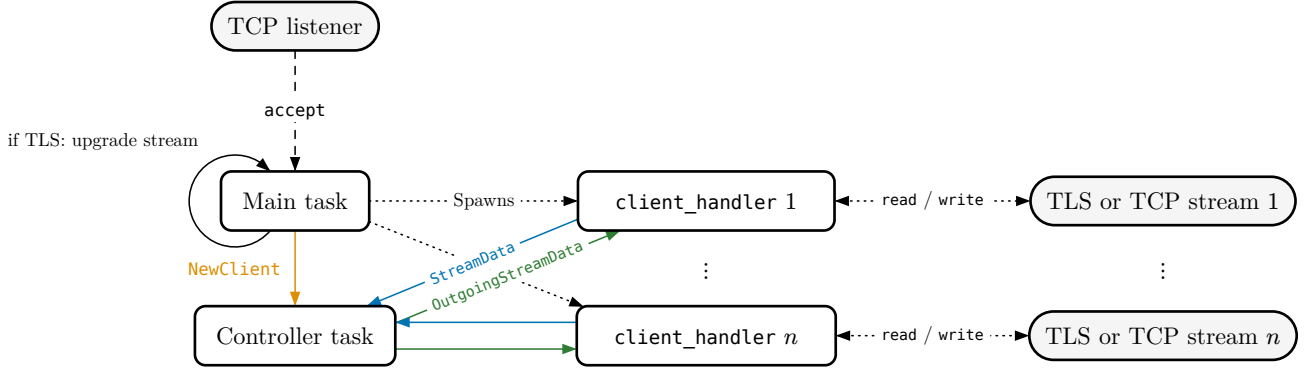


Figure 10: TCP server application architecture

QUIC uses TLS 1.3 for all connections. To measure the impact of encryption, TLS 1.3 support was added to the TCP application as an optional program flag. When disabled, the TCP stream obtained after accepting a connection is used directly. When enabled, the stream is upgraded to a TLS stream before being passed to the handler.

3.5.2 Performance tuning

The TCP baseline was optimized and configured to reduce protocol induced latency so that the comparison with QUIC-based implementations highlights the transport-level differences instead of default configuration parameters.

Nagle’s Algorithm. By default, TCP uses Nagle’s algorithm [43] to buffer small writes into fewer, larger packets. This reduces transmission overhead but adds latency because the sender waits either for enough data or for an ACK before transmitting its buffered data. For the evaluation of this application, where each message is sent using a write operation, it may have increased latency depending on transmission speed. Raising the `TCP_NODELAY` flag allows us to disable Nagle’s algorithm. This flag was raised on all sockets (both on the server side accepted streams and on the client streams).

TCP Delayed Acknowledgment. TCP Delayed Acknowledgments (RFC 1122, section 4.2.3.2) [44] delay the transmission of acknowledgments for up to 40ms on Linux [45] (or higher, depending on the kernel’s HZ setting) or until a second full-size segment is received. This reduces the number of ACKs but can interact with loss recovery. This mechanism can be disabled by setting the `TCP_QUICKACK` socket option [46] but this flag is not permanent and it must be raised after every ACK reception to reenter quick-ack mode.

In our evaluation, most tests did not introduce loss, and at a higher throughput the additional ACKs might have induced additional load on the server. Quick-ack mode was therefore left disabled. The corresponding QUIC

parameter `max_ack_delay` was kept at its default 25ms for all QUIC-based implementations.

TCP Slow Start After Idle. TCP Slow Start After Idle (SSAI, RFC 2861 [47] and 7661 [48]) resets the congestion window when a connection has been idle for at least one retransmission timeout (RTO). Linux defaults to a minimum RTO of 200ms (Kernel HZ divided by 5 [49]). At low throughput ($\lambda = 10$ packets per second, mean interval $\mu = \frac{1}{\lambda}$ second = 100 ms), roughly 13.5% of inter-packet gaps are longer than the 200ms RTO ($P(\text{gap} > 200 \text{ ms}) = e^{-200/\mu} = e^{-200/100} = e^{-2} \approx 0.135$). These gaps can cause the congestion window to reset to its initial value, thereby hurting tail latency. SSAI was disabled using `sysctl`. At higher throughput, the probability of the inter packet interval being larger than 200ms becomes exceedingly small (at $\lambda = 10000$ packets per second $\rightarrow \mu = 0.1$ ms and $P(\text{gap} > 200 \text{ ms}) = e^{-\frac{200}{0.1}} \approx 0$), rendering this setting optional.

Congestion control. The congestion control algorithm was set to CUBIC to match the QUIC configuration.

3.5.3 Rust Implementation

TLS library. `rustls` was selected as the TLS library for its integration with `tokio`'s TCP listener. TLS parameters were aligned with `quiche`'s defaults as much as possible. Both use `TLS_AES_128_GCM_SHA256` (AES with 128bit key in Galois/Counter mode) as the chosen cipher. Explicit flushes were added after each write to make sure that the TLS records were not buffered by `rustls`.

Memory handling. Vectors used in the message handling path were replaced with `BytesMut` instances. `BytesMut` is a reference counted shared buffer abstraction that can be frozen to avoid copies. This was applied to all application implementations.

3.6 Evaluation

This section evaluates the chat application implementations described above. The goal is to compare the end-to-end latency, downstream and upstream goodput, and CPU utilization of Flexicast QUIC against the `tokio-quiche` and TCP (with and without TLS) baselines, across topologies of varying size, varying network loss, and varying number of receivers.

3.6.1 Testing environment

All results presented in this section were obtained using Grid'5000 [50]. A single server from the `larochette` cluster in Luxembourg was used (`larochette-4`). This server is a Dell PowerEdge R760XA, has two Intel Xeon Platinum 8568Y+ (48 cores per CPU, for a total of 96 cores) and 512 GiB of memory. It is equipped with a 3.1 TiB Dell NVMe SSD. All files used

for the evaluation were located in the `/tmp` directory, on Grid’5000 servers, this directory is guaranteed to use the local disk [51], ensuring no interaction with NFS (Network File System) that may yield slower or inconsistent I/O operations. The operating system used is Debian 13 (`debian13-nfs` flavor), with Linux kernel version `6.12.86-1` (2026-05-08).

The application implementations run on virtual topologies running in Linux network namespaces. Each node defined in a topology (client, server, or router) runs its process in a dedicated namespace, and virtual ethernet pairs link the namespaces according to the topology’s links. The routers run FRRouting version 10.4.4 [52]. IS-IS [53] is used as the routing protocol. All routers are in the same IS-IS *area*, and are all operating at *level 1*. Protocol Independent Multicast (PIM) is used for multicast routing. More specifically, PIM-SSM is used given that our Flexicast QUIC fork uses SSM. IGMPv3 [14] is used for multicast group membership management.

The original topology scripts were authored by Anthony Doeraene for his Master Thesis “*Design of a multicast congestion control algorithm for Multicast QUIC*” (2025 [54], [55]). For this work, the scripts were extended to be able to create more diverse topologies based on YAML configuration files, which follow a structure similar to *ContainerLab*’s configuration files [56]. Generic Routing Encapsulation (GRE - RFC 2784 [17]) support was also added to test Flexicast QUIC over networks without full multicast support, see §4.5 for more information on GRE tunneling. The full configuration structure, default options, and an example topology file are provided in annex §A.3.

Tests are performed with NPF (Network Performance Framework [36]).

3.6.2 Testing methodology

Throughout each test, the sender (`client1`) continuously sends messages with an inter-packet interval drawn from a Poisson point process. Each message carries the current `Instant` converted to an unsigned 64-bit integer and a fixed-size random payload (§3.2). Every client (`client1...clientN`) receives the messages, parses the `u64` into an `Instant` (valid only when all processes run on the same machine), and computes the elapsed time in microseconds since the received timestamp. Every tested combination was run 5 times.

3.6.3 Topologies

Several virtual topologies were created using the aforementioned topology creation scripts. Each was designed to evaluate Flexicast QUIC in different conditions, and compare the baseline implementations in the same conditions. The diagrams below were automatically generated by the topology scripts using `graphviz`.

3.6.3.1 Solo: One client, one router, no delay

The first and smallest topology is designed to evaluate the implementation overhead of each version. In it, the sender (`client1`, also a receiver) is connected to the same router as the server with no delay placed on the links.

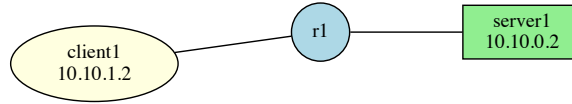


Figure 11: Solo topology

One might expect to observe Flexicast QUIC's latency to be equal or worse than `tokio-quiche`, and likely worse than TCP given that multicast provides no advantage over unicast transmission.

3.6.3.2 Tiny: Two clients, one router, no delay

Increasing the client count to two will let us observe if Flexicast QUIC gains a performance advantage as it must only encrypt and send each message once, compared to twice for the other applications. The clients and the server are connected to the same router.

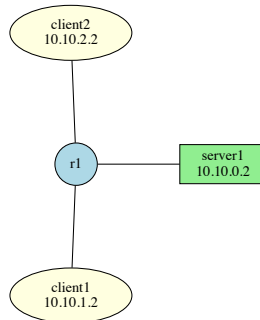


Figure 12: Tiny topology

3.6.3.3 Small: Three clients, three routers, no delay

The third topology adds two routers and one additional client. The server and the sender (`client 1`) are connected to the same router, while the other clients are connected to their own routers. The goal with the addition of two routers was to observe the impact on latency for multicast (if any).

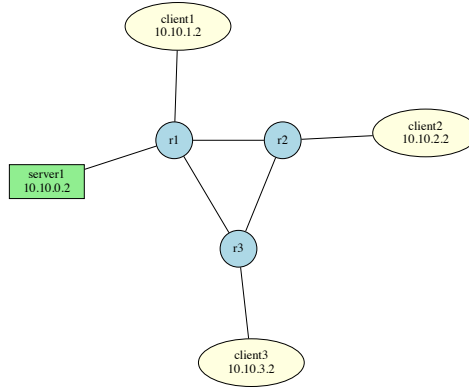
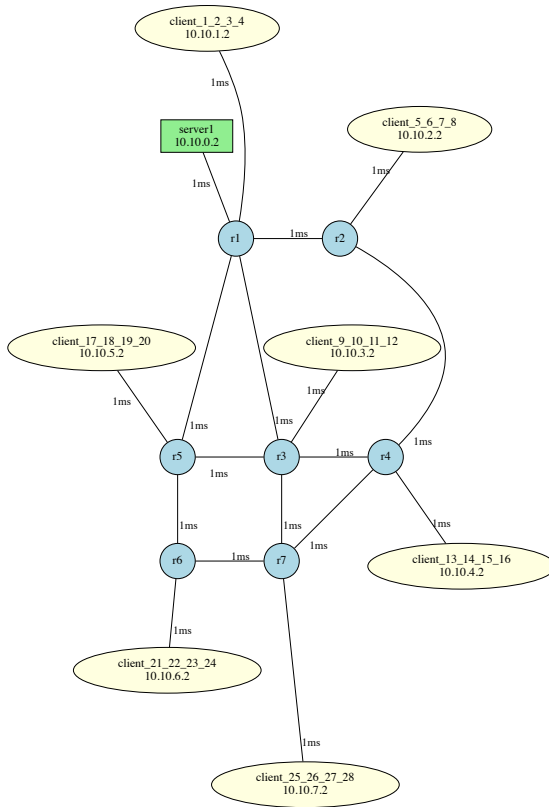


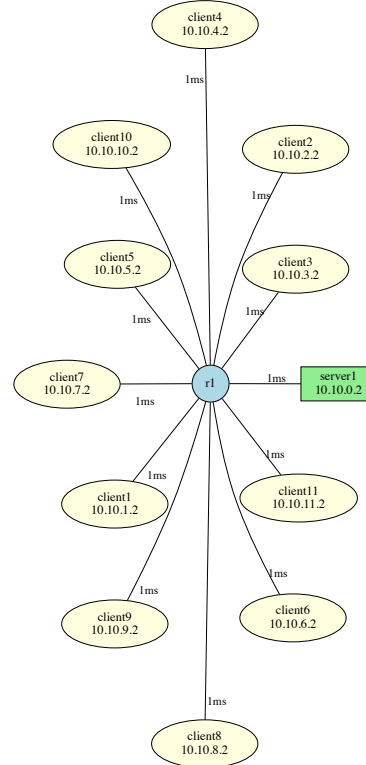
Figure 13: Small topology

3.6.3.4 Medium: 28 clients, seven routers, 1ms delay per link

Figure 14a presents this medium topology, which adds more clients, more routers and more links. Each link has one millisecond of delay. Note that, for readability of the diagram, the clients are merged together.



(a) Medium topology



(b) Data and Receivers test topology (subset of 11 clients)

Figure 14: Larger topologies

3.6.3.5 Receivers and Data Test Topologies

In order to test how Flexicast QUIC scales with an increasing number of receivers, or increasing message size, a different kind of topology was created. This topology has only 1 router with all clients connecting to it.

Figure 14b illustrates this topology with only 11 total clients for clarity. For the receivers test (§3.7.2), the total number of clients is 81 (including client 1), and for the Payload Sensitivity test (§3.7.3), it is 91.

3.7 Results

In this section, we look at three different categories of network conditions, client behavior and count, and we analyze the results.

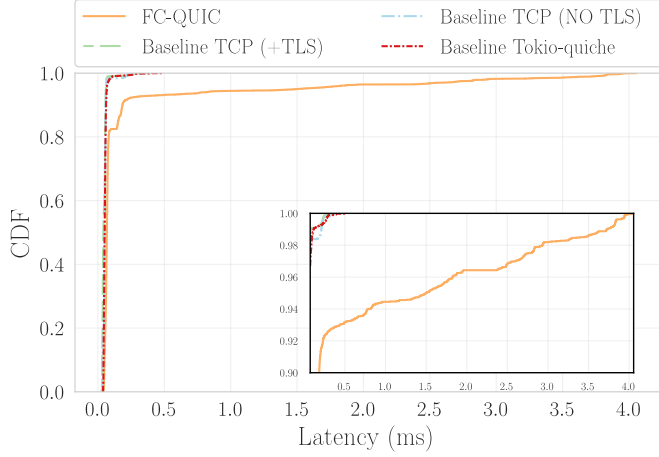
3.7.1 Network condition tests

The goal of this test is to evaluate Flexicast QUIC with topologies of varying sizes, with added latency and packet loss. We start with the Solo topology to isolate and measure the implementation overhead, then evaluate FCQUIC on the Tiny, Small, and Medium topologies to progressively introduce more receivers, routers, links (with and without delays), as well as loss rates.

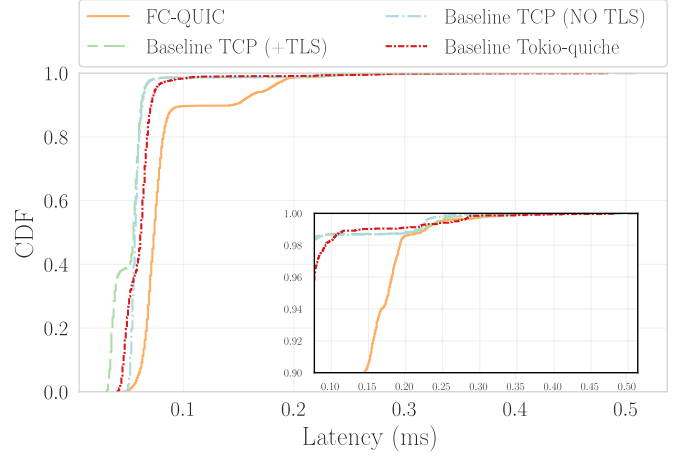
Solo topology: measuring implementation overhead. In the smallest possible network (Figure 11) with one client, one router, and one server, all server applications are expected to perform the same amount of work. That is, receiving messages from the only client, and sending the data back to the same client. Flexicast QUIC’s ability to encrypt and send the message only once will not be an advantage in this scenario, in fact, the complex nature of the implementation will likely hurt its performance.

The sender’s outgoing rate is 500 messages per second. The payload size for this test varied from 0 to 1100 bytes (with 3 steps in between). With a 0-byte payload, only the client ID and timestamp are sent in the messages. Even at 1100 bytes, each message fits into one QUIC or TCP packet with no segmentation required. Note that the workload will be higher with a 1100-byte payload than with no payload. The Payload Sensitivity test (§3.7.3) will fix the bandwidth with regard to the increasing payload size.

The server was allocated two threads, pinned to cores 0 and 1. The client application is also allocated two threads.



(a) Payload = 0 bytes



(b) Payload = 1100 bytes

Figure 15: CDFs of latency in milliseconds for different payload sizes

Figure 15a and Figure 15b present the CDFs for 0-byte and 1100-byte payloads, respectively. In both cases, the baselines have nearly vertical CDFs below 0.1 ms, while FCQUIC’s CDF tends to rise a little slower, indicating a larger variance.

With a 0-byte payload (Figure 15a), the latency gap is very pronounced in the long tail, FCQUIC only converges to 100% at 4 ms, whereas all baselines converge at 0.5 ms. FCQUIC’s p. 99 is 3.72 ms and is roughly 48 times higher than TCP+TLS’s p. 99 latency of 0.077 ms, however, the median is only 1.7 times higher (0.068 ms vs 0.041 ms). This high tail latency at a small message size may come from FCQUIC’s architecture. With a 1100-byte payload (Figure 15b), the latency gap shrinks significantly: FCQUIC’s CDF is still trailing the baselines, but FCQUIC reaches the 99th percentile at 0.226 ms, basically identical to the baselines.

FCQUIC’s implementation overhead is clearly visible, as the message size increases, so does the workload, yet FCQUIC’s p. 99 latency improved, possibly resulting from the heavy use of message passing internally. It must be noted that the implementation is far from being overloaded, with a mean CPU utilization percentage at 54%, as shown in Figure 16. FCQUIC’s architectural complexity is reflected in its high CPU usage, which is roughly 135 times higher than TCP+TLS’s mean utilization of 0.4%.

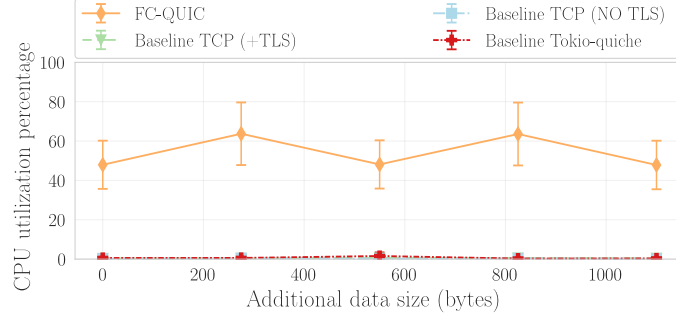


Figure 16: Mean CPU utilization percentage for each payload size

Figure 17 presents the goodput measured by the server based on the message size transmitted to the receivers. As expected in this topology, the goodputs of all implementations are identical and grow linearly as the message size increases, reflecting the increased workload caused by larger messages sent at the same interval.

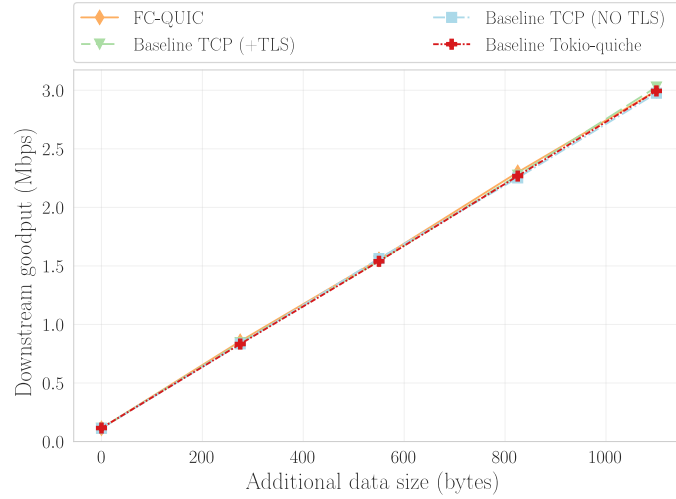


Figure 17: Mean downstream (towards clients) goodput in Mbit/s

Tiny topology: comparing implementations with two receivers. The second evaluated network is a topology with two clients, still connected to the same router as the server (Figure 12). As before, the sender transmits messages at a rate of 500 messages per second, and all messages carry a 1100-byte payload (on top of the timestamp and client ID).

The upstream goodput (application-level throughput) received by the source reflects the message transmission rate and is identical for all implementations, as shown in Figure 18.

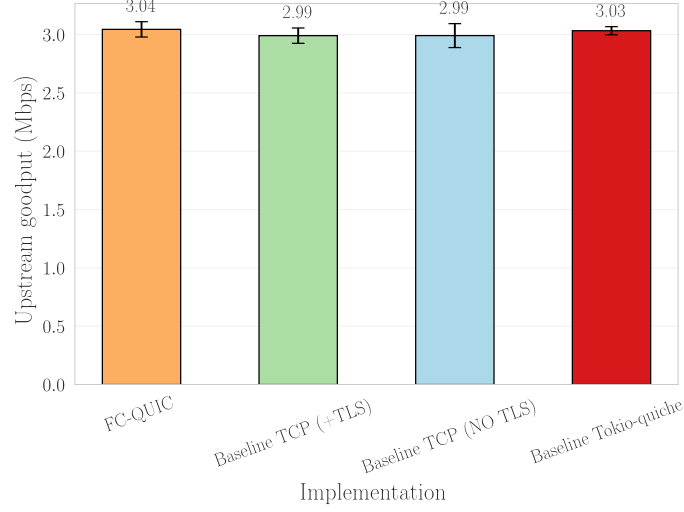


Figure 18: Mean upstream (towards source) goodput in Mbit/s

When distributing packets using unicast to two clients, we expect the downstream (towards the clients) goodput to be double that of multicast. This is directly visible in Figure 19, where FCQUIC’s downstream goodput is half that of the baselines.

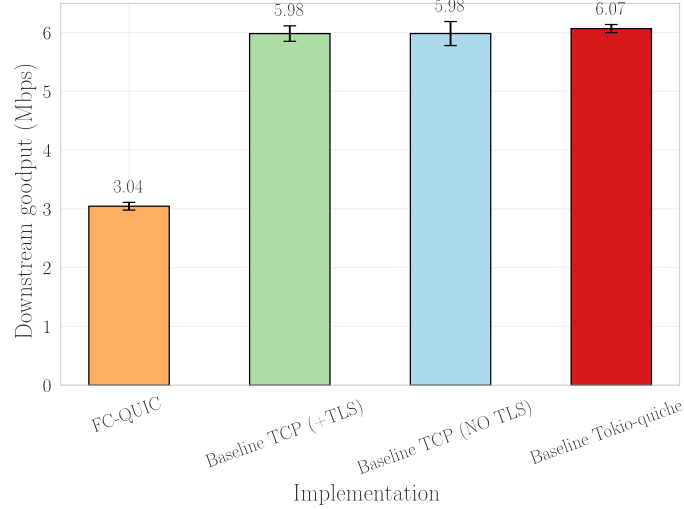


Figure 19: Mean downstream (towards clients) goodput in Mbit/s

Figure 20 presents the latency CDF, highlighting a long tail latency for FCQUIC with a p. 99 latency of 3.89ms compared to 0.81 ms for TCP+TLS and 0.67 ms for `tokio-quiche`, roughly 4.7 times higher and 5.7 times higher, respectively, showing once again the architectural complexity’s cost. FCQUIC’s median latency, however, is competitive with the baselines. The mean CPU utilization is 55.6% for FCQUIC, compared to 0.69%, 0.99%, and 0.4% for TCP+TLS, TCP, and `tokio-quiche` respectively.

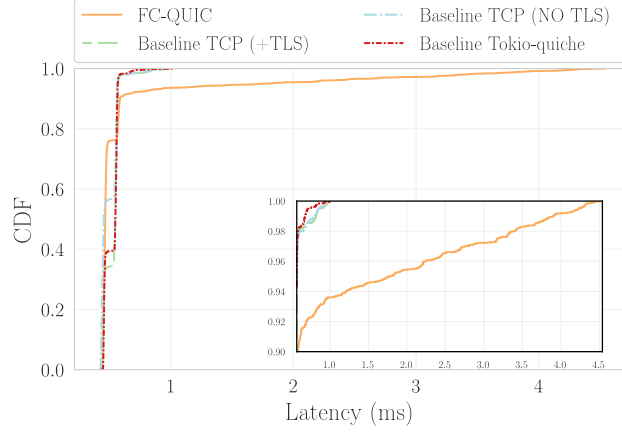


Figure 20: CDF of latency in ms

Small topology: with and without loss. In a slightly larger network with 3 clients and 3 routers, we evaluate Flexicast QUIC’s response to different amounts of loss placed on each link. For this test, four variants of the topology were created, with 0%, 0.5%, 5%, and 10% loss (all with 25% burst loss chance) set on each link.

As before, the sender is transmitting 500 messages per second, and all messages carry a 1100-byte payload (on top of the timestamp and client ID).

With 3 clients, each baseline will send three copies of every message, while FCQUIC will send one. This result is presented in Figure 21. Do note, however, that this value should not increase with high network loss. That is due to the goodput measurements being taken at the application level, measuring how many bytes are sent. Lost packets will be retransmitted by the transport and, therefore, will not be represented in this graph.

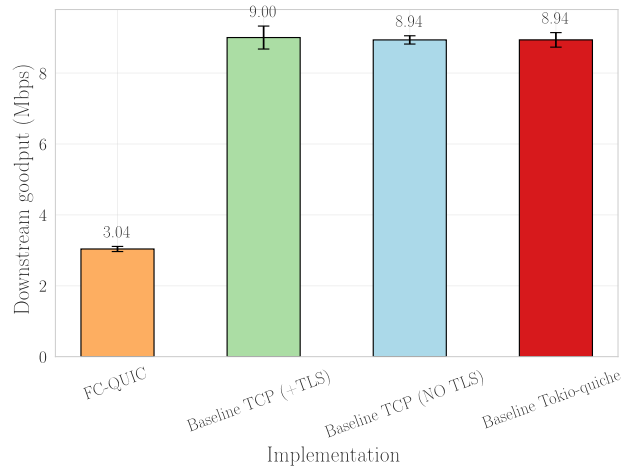
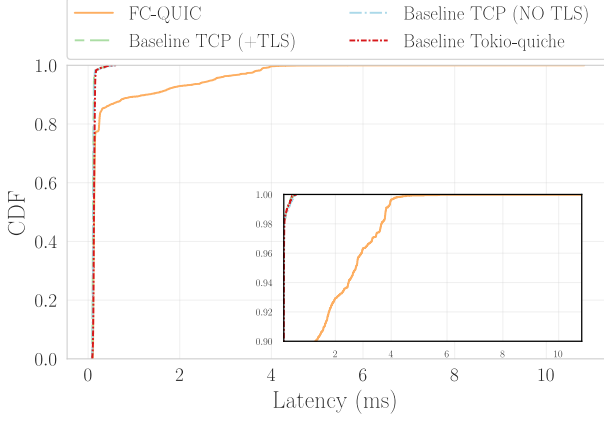
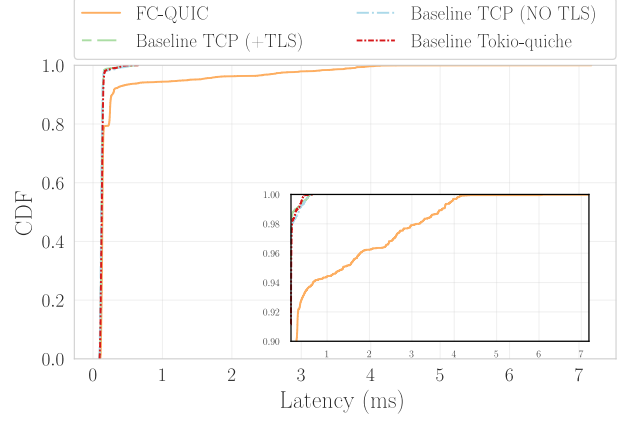


Figure 21: Mean downstream (towards clients) goodput in Mbit/s with 0% loss

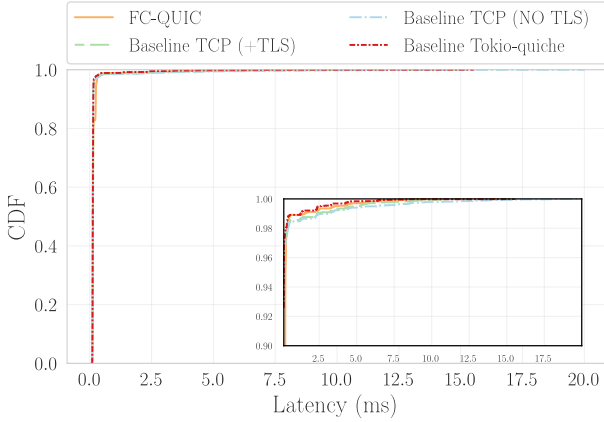
Zero percent network loss is highly desirable but rare. 0.5%, however, is closer to real network loss levels in larger ISP networks [57]. Five percent and 10% loss rates, while much higher, may better reflect worst-case losses using satellite internet providers such as Starlink [58]. These values were chosen to evaluate FCQUIC’s behavior under packet loss, but to also evaluate if FCQUIC’s behavior under loss is different due to its own loss recovery mechanism, or due to QUIC’s differences with TCP.



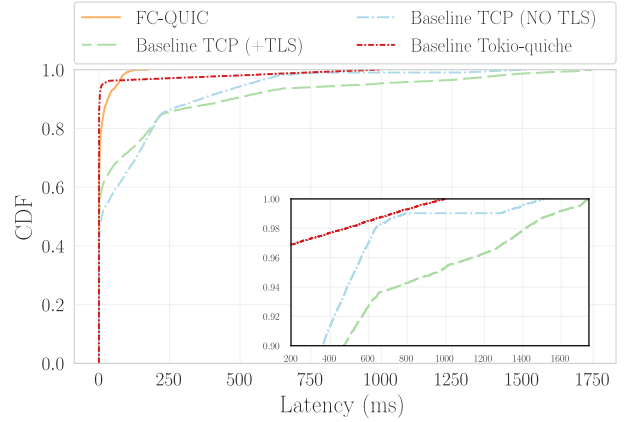
(a) 0% loss



(b) 0.5% loss



(c) 5% loss



(d) 10% loss

Figure 22: CDF of latency in milliseconds with and without loss

Under no loss, in Figure 22a, the CDF shape is similar to that of the previous tests, showing FCQUIC’s high tail latency. The behavior is very similar at 0.5% loss, in Figure 22b, but FCQUIC’s CDF now rises faster, reaching a p. 95 latency of 1.3ms under 0.5% loss, compared to p. 95 of 2.75ms under 0% loss. Its p. 99 latency remains very high at 3.7ms for 0.5% loss compared to 3.8ms for 0% loss.

Higher network loss of 5%, in Figure 22c, starts to show real differences. All implementations now have a very similar CDF shape, which rises fast

to the 98th percentile at roughly 1.5ms. However, the p. 99 latency of the QUIC-based implementations is lower than the TCP baselines, with 1.51ms and 1.26ms for FCQUIC and `tokio-quiche` respectively, while TCP baselines have 2.36 ms and 3.1 ms respectively (TCP+TLS and TCP without TLS).

Ten percent loss, as presented in Figure 22d, shows a large latency impact on the TCP baselines, with their p. 90 latency reaching 469 ms (for TCP+TLS) and 360 ms (TCP), while FCQUIC’s p. 90 latency is 31 ms and `tokio-quiche`’s is 2.44 ms. Here, both QUIC-based implementations obtained low latency, with a CDF that rises sharply compared to the TCP baselines, whose CDF started to rise at a shallower angle after the 50th percentile. Despite `tokio-quiche`’s good p. 90 latency, its p. 99 latency is very high, as seen in the long tail reaching 723 ms. It is 7.5 times higher than FCQUIC’s 96 ms of p. 99 latency. The TCP+TLS baseline has a p. 99 latency of 1730 ms.

QUIC-based server and client implementations send every message inside a new stream. If part of a stream is lost, this does not stop any other streams from progressing. This behavior is clearly visible on the CDF: a sharp rise in the CDF indicating the majority of streams arrived quickly, with a long tail consisting of retransmitted stream frames arriving later. The TCP implementations are suffering from head-of-line blocking, leading to the shallower shape of the CDF.

Medium topology: larger network with delays and loss. The medium topology (Figure 14a) has delays placed on the links, a lot more routers, and 28 clients. This simulates a larger network than before.

As before, the sender is transmitting 500 messages per second. All messages carry a 1100-byte payload, and the server and clients are pinned to two threads.

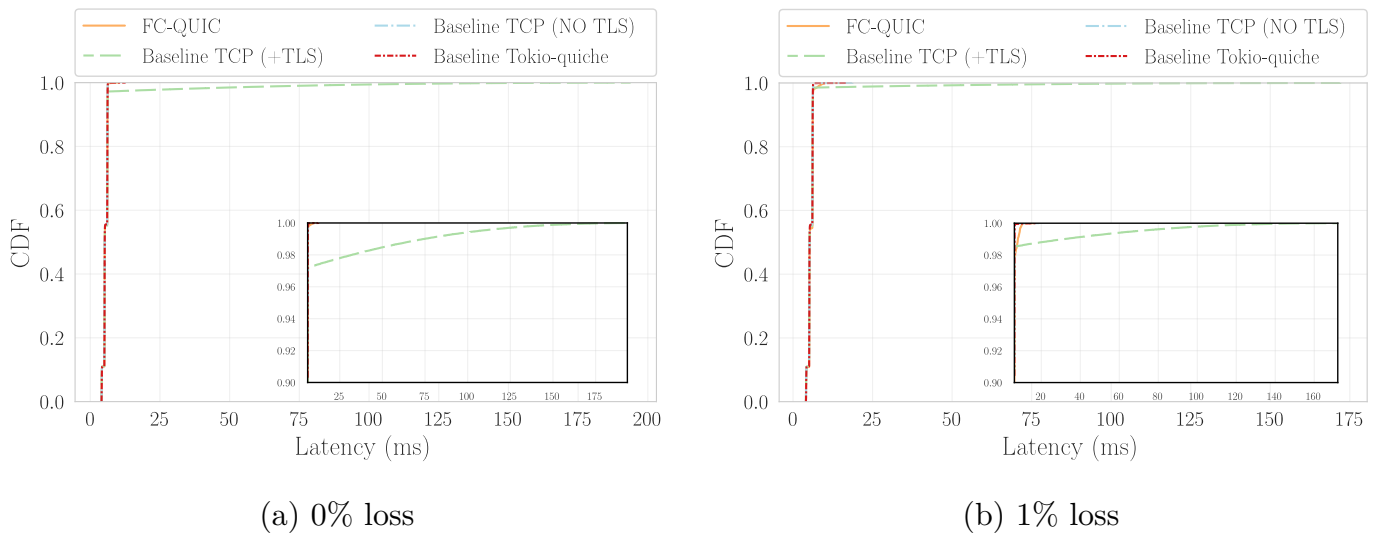
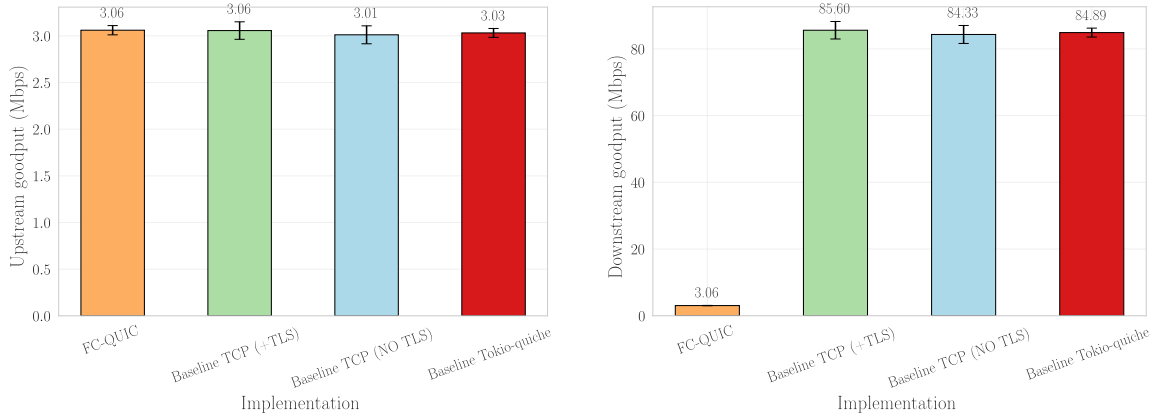


Figure 23: CDF of latency in milliseconds with and without loss

Figure 23 presents the CDFs of the implementations under 0% loss for Figure 23a and 1% for Figure 23b. Both figures show the fast rising CDF of all implementations, with a long tail in the hundreds of milliseconds for baseline TCP+TLS. Under 0% loss, the mean latency of FCQUIC is 5.49ms, compared to 7.10 ms for TCP+TLS (with a 95% CI of 0.067 ms), 5.4ms for TCP, and 5.4ms for `tokio-quiche`. The p. 99 latency for FCQUIC is 6.3ms, which is 11 times lower than TCP+TLS’s p. 99 latency of 74.2 ms. Under no loss, TCP+TLS has a high tail latency serving 28 clients. This same latency is not observed in the other baselines.

Under 1% loss, the results remain largely the same. FCQUIC’s mean latency is 5.5ms, TCP+TLS is 6.21ms, TCP is 5.46 ms, and `tokio-quiche` is 5.48 ms. With higher loss, however, TCP+TLS’s p. 99 latency is not as high as with 0% loss, reaching 31.3 ms, while FCQUIC reaches 8 ms. The CPU utilization remains as before across all implementations, with FCQUIC’s mean utilization staying at 54%, with TCP+TLS at 2.9%, TCP at 1.75%, and `tokio-quiche` at 2.76%. The work added by encryption is clear in the 1.15% difference between TCP+TLS and TCP.



(a) Mean upstream goodput (towards the source) at 0% loss (b) Mean downstream goodput (towards the clients) at 0% loss

Figure 24: Up and downstream goodputs on the medium topology with 0% loss

Figure 24 presents the upstream goodput in Figure 24a and downstream in Figure 24b, clearly showcasing the added work performed by the baselines compared to the FCQUIC source. It must be noted that this workload is not yet overloading the sources. Setting the transmission rate higher will result in sources eventually becoming overloaded, but may not be representative of a lighter workload, such as a chat application or stock ticker updates.

3.7.2 Receivers Scalability

One of the main advantages of multicast distribution is its ability to scale and accommodate many receivers. The goal of this test is to assess Flexicast

QUIC’s scalability with respect to the number of receivers, and to compare it against the baselines as the receiver count grows. The topology used is the one presented in Figure 14b, and we vary the number of receivers from 2 up to 81, with no network loss.

The transmission rate from the sender is 500 messages per second. Again, the servers are allocated and pinned to two cores, and the core utilization percentage is recorded at a regular interval.

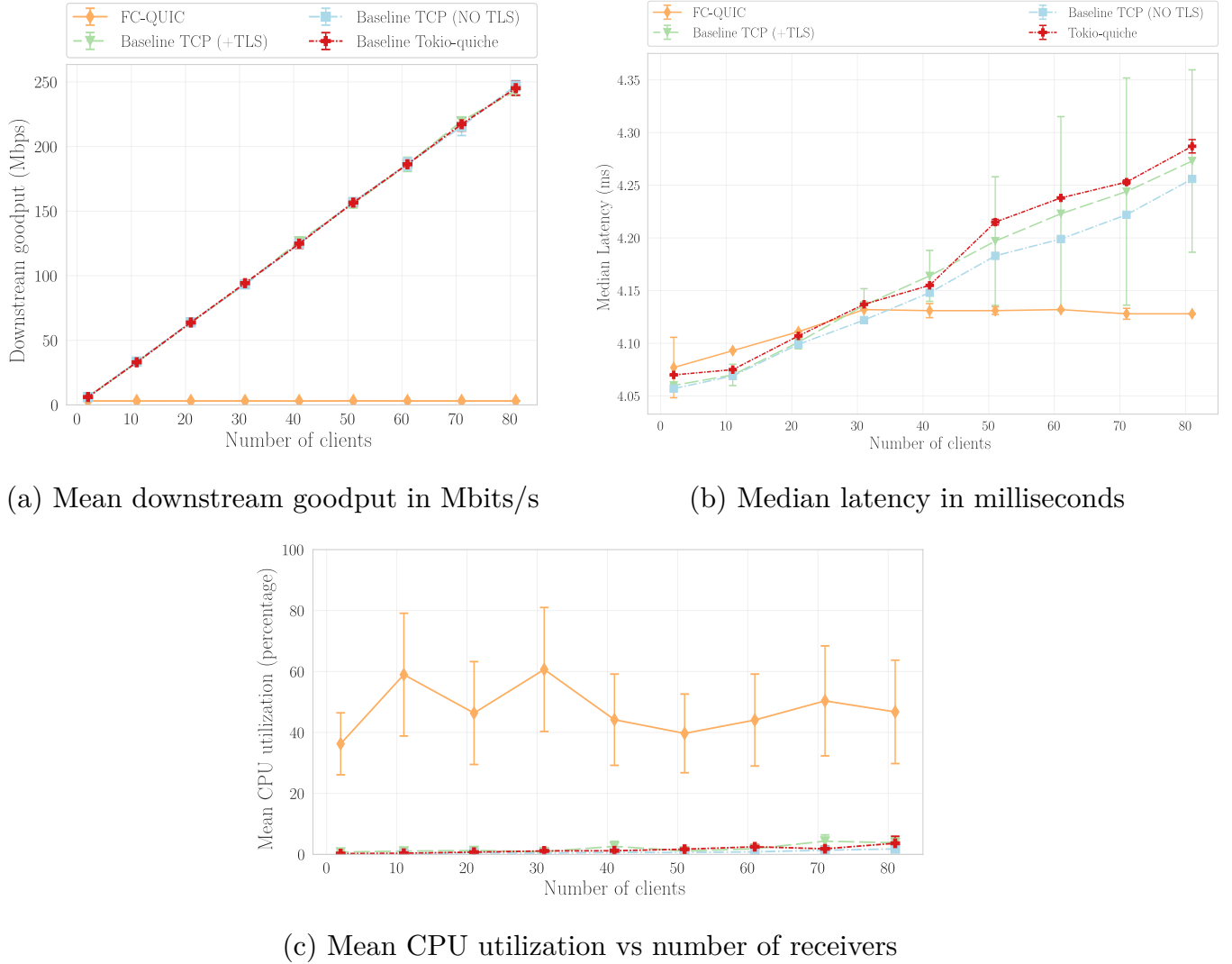


Figure 25: Scalability of implementations w.r.t. number of receivers

Figure 25 presents several interesting graphs from this test. First, in Figure 25a we observe the downstream goodput rising linearly with the number of clients for the baseline implementations, while remaining constant for FCQUIC. This matches the earlier observations, and does indicate that, even at 81 receivers, the baseline implementations are able to keep up with the workload, otherwise a decrease in the slope would be visible.

Figure 25b presents the median latency of the implementations. FCQUIC’s median latency rises initially from 4.07ms at 2 receivers to 4.13ms at 31 receivers and remains stable at 4.12 to 4.13 ms. The other implementations followed the same initial rise in latency, except their median latency continued to climb past 31 receivers, going up to 4.27ms at 81 receivers for TCP+TLS and 4.33 ms for `tokio-quiche`. This median latency increase as the number of receivers increases is also visible in Figure 25c, where the baselines see their mean CPU usage rise slightly past 31 receivers.

Figure 26 presents the CDF of the latency when 81 receivers are present. The figure allows us to see that FCQUIC’s p. 99 latency of 4.3 ms is on par with most baselines (5.2 ms for `tokio-quiche` and 4.7 ms for TCP), except TCP+TLS whose p. 99 latency is 117.7 ms, roughly 27 times higher than FCQUIC.

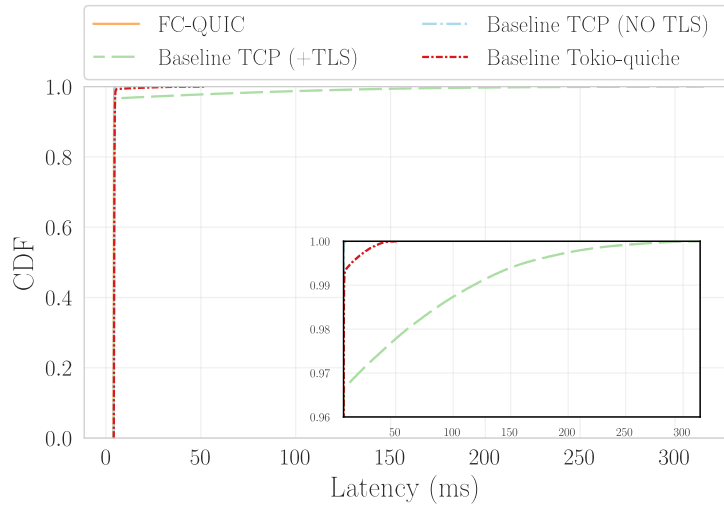


Figure 26: Latency CDF with 81 receivers

3.7.3 Payload sensitivity

The goal of this test is to observe the effect of increasing message size on the implementations while keeping the upstream goodput roughly constant. To do so, we adjust the transmission rate as the message size grows. As the messages grow larger, more stream frames or TCP segments are required to transmit each message, and the receivers only compute the latency once the entire stream, or the complete TCP message, has been reassembled and sent up to the application. The topology used is the one presented in Figure 14b, with 91 receivers.

1. *Small messages:*

- Payload size = 1100 bytes, transmission rate = 500 messages per second

2. *Medium messages:*

- Payload size = 2200 bytes, transmission rate = 260 messages per second

3. *Large messages*:
 - Payload size = 3500 bytes, transmission rate = 167 messages per second
4. *Huge messages*:
 - Payload size = 4700 bytes, transmission rate = 125 messages per second
5. *Extra large messages*:
 - Payload size = 6500 bytes, transmission rate = 90 messages per second
6. *Jumbo messages*:
 - Payload size = 9000 bytes, transmission rate = 66 messages per second

The goal is to observe the impact of segmentation (splitting message content into multiple segments or stream frames when larger than the packet size), but also observe the impact of encryption of large messages. As before, the server is allocated and pinned to two CPU cores. Ninety-one receivers are used in this test.

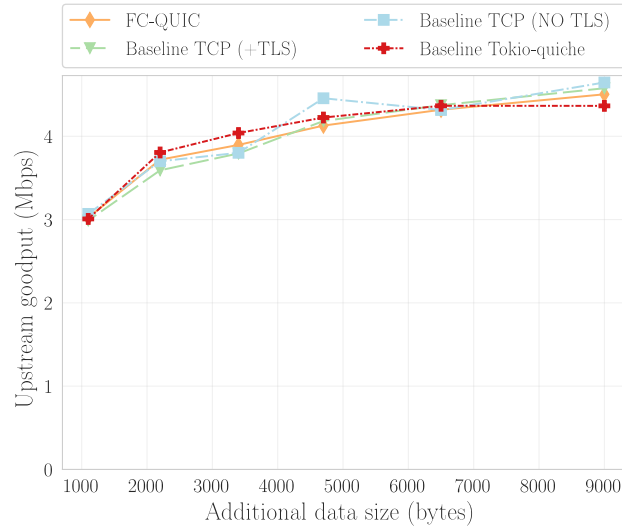
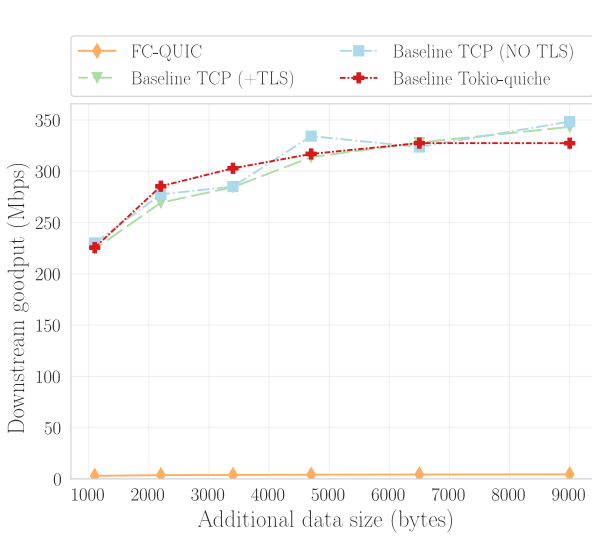
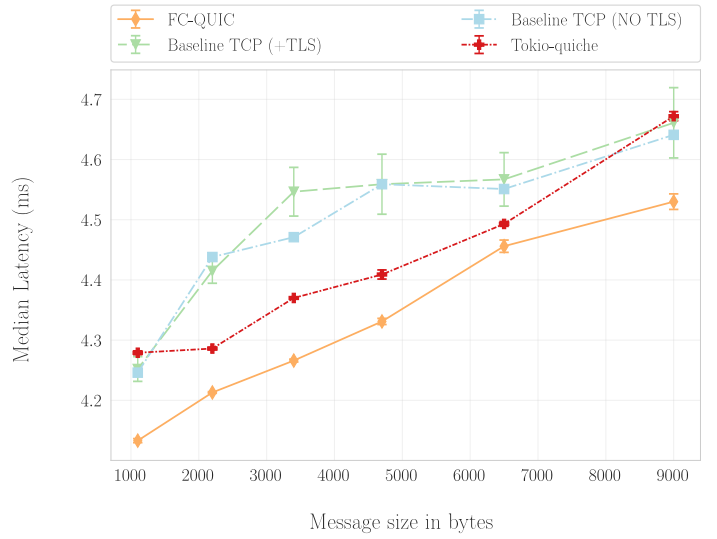


Figure 27: Mean upstream goodput (towards the source)

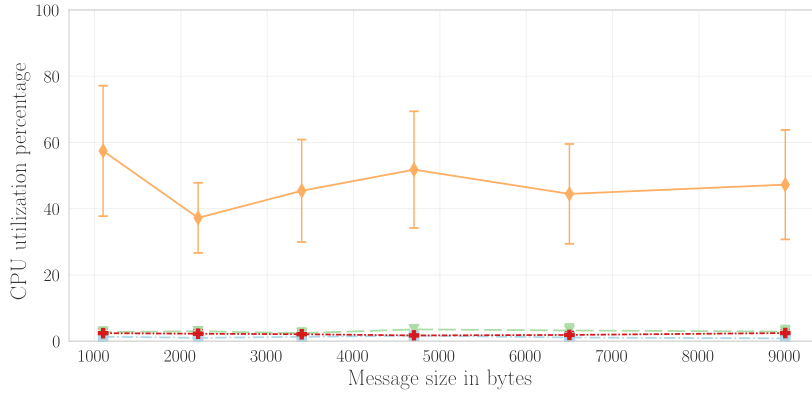
Figure 27 presents the upstream goodput. It increases slightly as the message size grows, from 3 Mbits/s for 1100-byte payloads to 4 Mbits/s for 4700-byte payloads, reaching roughly 4.5 Mbits/s for 9000-byte payloads. This increase is consistent across all implementations, and suggests that, at higher transmission rates, the sender’s transmission loop is too imprecise or unable to sustain the message transmission rate. Once the rate is low enough, the upstream goodput matches a rough throughput estimate of 4.75 Mbits/s, obtained by multiplying the message size (excluding protocol overhead) in bytes by 8 and by the transmission rate: $9000 \times 8 \times \frac{66}{10^6} \approx 4.75$ Mbits/s.



(a) Mean of downstream goodput in Mbits/s



(b) Median latency in milliseconds



(c) CPU utilization vs payload size

Figure 28: Payload Sensitivity results

The downstream goodput of the baselines, shown in Figure 28a, follows the same increase as the upstream goodput. This does not mean that the server is unable to maintain this rate, however, it is a result of the client's upstream rate increasing. FCQUIC's goodput remains constant as before. Figure 28b presents the median latency, which shows a steady increase for all implementations, with the TCP baselines' latency being higher than the QUIC-based implementations. TCP+TLS's mean latency across all message sizes is 7.7ms, higher than FCQUIC's mean 4.3ms. We notice again TCP+TLS's high tail latency if we look at Figure 29 which shows the CDF of the latency when sending messages with a 9000-byte payload. TCP+TLS's p. 99 latency in that scenario reaches 187.4 ms, 23 times higher than FCQUIC's p. 99 latency of 8 ms. TCP's p. 99 latency is 5.3ms, while tokio-quiche's is 5.4ms.

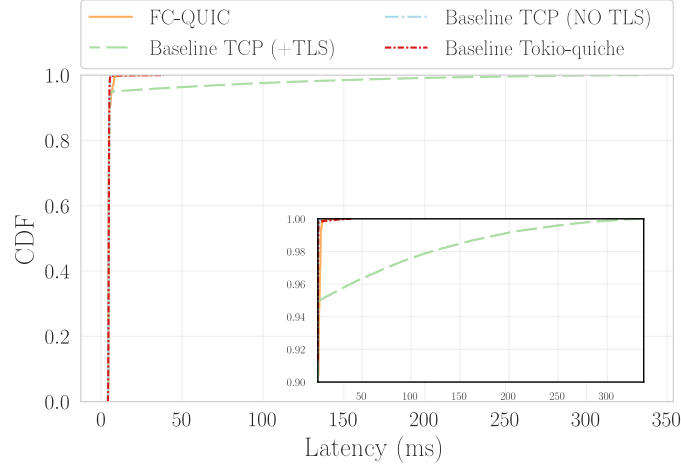


Figure 29: Latency CDF with payload size = 9000 bytes

Figure 28c presents the CPU utilization for the implementations. Unsurprisingly, FCQUIC’s mean utilization percentage is higher than the baselines, hovering at a mean of 47%, well above TCP+TLS’s mean of 2.9%. A very slight increase in CPU utilization is visible for TCP+TLS as the payload size increases.

3.8 Conclusion

This chapter evaluated a multicast chat application implemented using Flexicast QUIC, QUIC (with `tokio-quiche`), and TCP (with and without TLS) on virtual network topologies of increasing complexity.

The implementation overhead was first observed and measured on the solo topology, with one client, one server, and one router. With no payload in the messages, FCQUIC’s p. 99 latency was 3.72 ms, roughly 48 times higher than TCP+TLS’s p. 99 of 0.077 ms, while its median latency was only 1.7 times higher (0.068 ms compared to 0.041 ms). Increasing the payload to 1100 bytes lowered FCQUIC’s p. 99 down to 0.226 ms, virtually identical to the baselines. The shrinking latency gap, paired with a mean CPU utilization of 54% for FCQUIC compared to 0.4% for TCP+TLS, tends to confirm that the latency gap on this topology comes from FCQUIC’s implementation complexity.

By adding a second receiver in the tiny topology, the multicast advantage started to appear, with FCQUIC’s downstream goodput being half that of the baselines, due to each message being encrypted and transmitted only once on the multicast flow. The latency, however, was still high due to implementation overhead, with FCQUIC’s p. 99 reaching 3.89 ms against 0.67 ms for `tokio-quiche` and 0.81 ms for TCP+TLS.

The small topology added a third client and two routers, and evaluated each implementation under 0%, 0.5%, 5%, and 10% network link loss. Under low loss, the results were similar to the previous topologies. At 5% loss, all CDFs converged to have a similar shape, but the QUIC-based implementations maintained a lower p. 99 (1.51 ms for FCQUIC and 1.26 ms for `tokio-quiche`) compared to the TCP baselines (2.36 ms for TCP+TLS, 3.1 ms for TCP). Under 10% loss, the TCP baselines suffered a lot due to head-of-line blocking, with a p. 90 latency of 469 ms (TCP+TLS) and 360 ms (TCP), while FCQUIC’s p. 90 stayed at 31 ms and `tokio-quiche`’s at 2.44 ms. The QUIC-based implementations avoided this issue by using independent streams to carry messages, allowing the receivers to process other streams while a lost one was retransmitted.

With the medium topology, the network size grew to include 28 clients, 7 routers, and 1 ms of delay per link. Under no loss, FCQUIC’s mean latency was 5.49 ms, very close to TCP’s 5.4 ms and `tokio-quiche`’s 5.4 ms, and actually lower than TCP+TLS’s 7.10 ms. The tail latency, however, was in favor of FCQUIC, whose p. 99 latency was 6.3 ms, which is 11 times lower than TCP+TLS’s 74.2 ms. The results were largely the same with a 1% loss. Mean CPU utilization stayed consistent as before, with FCQUIC at 54% and the baselines below 3%.

The receiver scalability test increased the receiver count from 2 to 81 in steps of 10. FCQUIC’s median latency stayed within 4.07 ms and 4.13 ms across all receiver counts, while the baselines’ latency increased past 31 receivers up to 4.27 ms for TCP+TLS and 4.33 ms for `tokio-quiche` at 81 receivers. The tail latency highlighted the differences in implementation even more: at 81 receivers, the p. 99 latency for TCP+TLS was 117.6 ms, which is roughly 27 times higher than FCQUIC’s 4.3 ms. The downstream goodput increased linearly with the number of receivers for the baselines, as is expected, while FCQUIC’s stayed constant, also expected given its use of multicast distribution.

The payload sensitivity test attempted to fix the throughput while increasing the payload per message from 1100 to 9000 bytes on a network topology with 91 receivers. The TCP baselines’ latency increased unsteadily with message size, fluctuating quite a bit, while the QUIC-based implementations remained steadier and lower. TCP+TLS’s mean latency was 7.7 ms across all message sizes, which is higher than FCQUIC’s 4.3ms. With a 9000-byte payload, TCP+TLS’s p. 99 latency reached 187.4 ms, 23 times higher than FCQUIC’s p. 99 latency of 8 ms, while TCP and `tokio-quiche` remained low at 5.3 ms and 5.4 ms, respectively.

Overall, FCQUIC’s implementation overhead is measurable on small topologies, and is reflected in consistently high CPU utilization across all tests. This overhead seems to have a fixed cost at low speeds such as the

ones used for this evaluation. This cost, however, is quickly absorbed by the latency gains, as the number of receivers, message size, or network loss increases.

Chapter 4

Flexicast QUIC Relay

In this chapter, we address the acknowledgment implosion problem in Flexicast QUIC without adding an acknowledgment delay. We first discuss existing solutions and how Flexicast QUIC currently avoids the problem, then propose the addition of *relay* nodes to create a hierarchical distribution tree. We describe the *implicit* relay design and present its implementation. Finally, we detail the evaluation setup and discuss the results. A second relay design, the *explicit* relay, is presented in §5 as a future evolution of this work.

4.1 Motivation: Acknowledgment implosion

Reliable multicast protocols, like Flexicast QUIC, are capable of supporting hundreds of receivers and reaching throughputs up to 80 Gbps on a single server [11]. However, the source’s workload increases with each new client due to acknowledgment handling and retransmissions. When a source distributes to N receivers on a multicast flow, each receiver will acknowledge the received packets. When N grows, this acknowledgment volume increases linearly and can eventually overwhelm the source due to the problem known as *acknowledgment implosion* [1], [59].

Prior work on reliable multicast protocols such as Reliable Multicast Transport Protocol (RMTP) [59] and Negative-ACKnowledgement Oriented Reliable Multicast (NORM) protocol [60] has avoided the problem in different ways. RMTP defines a hierarchical structure where receivers are grouped in local regions, with a Designated Receiver (DR) responsible for acknowledging data and handling retransmissions locally [59]. NORM, on the other hand, suppresses redundant acknowledgments by using an exponentially distributed random back off timer, but notably does not maintain per-receiver state, and can rely on multicast or unicast for feedback distribution [60]. This provides scalability but little flexibility.

Flexicast QUIC, as demonstrated in [11], can work around the problem by introducing an *acknowledgment delay*, which prevents the implementation from sending more than 1 acknowledgement per ack delay period (e.g., one acknowledgement per 10 milliseconds if the ack delay is 10 ms). This approach can reduce the rate of acknowledgement from ~1.4 Gbps with no ack delay to ~200 Mbps with an ack delay of 5ms [11] (with 1000 receivers and an 80 Mbps data stream sent by the source).

4.1.1 Proposed Solution: Hierarchical Relay Architecture

To address the acknowledgment implosion problem without adding an ack delay, we propose introducing relay nodes between the Flexicast QUIC source and the receivers. A relay acts both as a client of the FCQUIC source and as a source to downstream clients. We create a hierarchical distribution tree in which:

1. The FCQUIC source only handles ACKs from a few relays and potentially some directly connected clients.
2. Each relay manages its own multicast flow(s) and handles acknowledgments from downstream clients.
3. The relays can handle retransmissions directly by storing messages, this can reduce retransmission latency.

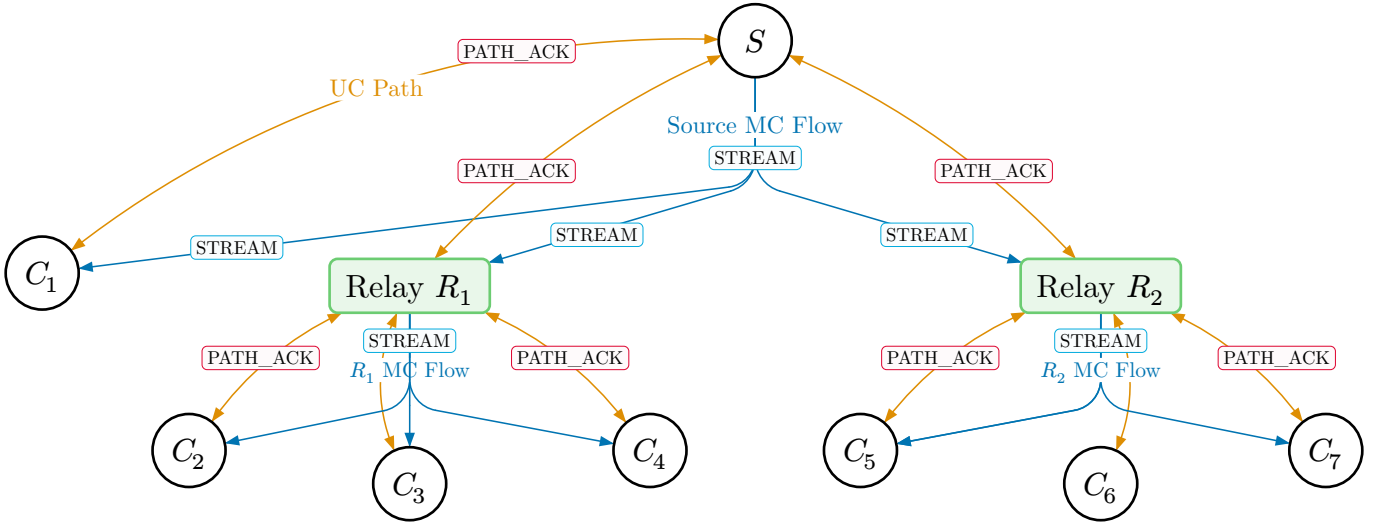


Figure 30: Hierarchical relay architecture

Figure 30 illustrates this scheme with a client C_1 directly connected to the source S . This source has two relays: R_1 and R_2 as clients, both of which have three downstream clients. Every client joined the multicast flow of their source (either the source S or a relay) over which they receive stream data. Every client maintains a unicast path towards their source, over which they send acknowledgments. The relays redistribute the source's stream data to their downstream clients and handle their acknowledgments.

The relay design is similar to RMTP's Designated Receivers [59], but also to Media-over-QUIC (MoQ) [61] relays. In MoQ, the relays terminate the MoQ session on both sides to have access to the MoQ object metadata. Additionally, the MoQ relays, like MoQ servers, are discovered out-of-band (using DNS, via the application, or with configuration values). MoQ relays may handle retransmissions locally if they cache objects [61].

4.2 Implicit Relay Design

The implicit (transparent) relay design makes the relay invisible/unknown to both the source and the clients at the protocol level. The relay connects to the FCQUIC source as a normal client, joins the multicast flow, then separately establishes itself as a new FCQUIC source with its own multicast flow. Clients connect to the relay as they would to a normal FCQUIC source.

Transparent relay operation. After connecting to the FCQUIC source and joining its multicast flow, the relay advertises its presence and role as a relay to potential clients through out-of-band mechanisms such as DNS configuration or application information. For example, in a live-streaming context where a client wants to watch a live event via a web application, the client will connect to whichever address was communicated by the backend service. This address may be the FCQUIC source, or it may be a relay.

The relay forwards messages received from the source’s multicast flow on its own multicast flow. Depending on the application, clients may or may not send messages to the source to be distributed over the multicast flow. If the application permits it, messages received on the unicast paths from the relay’s clients can be forwarded to the source using the relay’s own unicast path to the source (in which case messages should contain some identifying information, e.g., a client ID or client username).

4.2.1 Message Caching

Just like Media-over-QUIC relays, the FCQUIC relay may cache messages. We define two message caching modes:

Non-caching mode. The relay will forward the source’s messages directly on its own multicast flow. It aggregates ACKs from its clients, and sends an aggregated ACK message after a short period of time. If a client does not ACK a message, the relay will not include the message in the aggregated ACK to the source. It lets the source retransmit the message via the relay’s unicast (UC) path, and will transmit this message over the client’s UC path. This mode of operation allows for a less complex relay implementation but incurs higher retransmission latency. This mode of operation may be interesting in video streaming situations where not all packet losses are equally bad. For example, the loss of an I (intra) frame (which is a self-contained frame containing all the information required to display the frame) is worse for service quality than the loss of a P (predicted) frame (a frame encoding differences from the previous frame), which may already be too old by the time a retransmission arrives. Keeping retransmission decisions at the source allows the application to make decisions for each frame type and not require the relays to be aware of the semantics of each message it is forwarding.

Caching mode. The relay stores received messages until all downstream clients acknowledged the message. Caching relays acknowledge messages received from the source directly and handles retransmissions locally. This mode of operation significantly reduces the retransmission latency and allows the relays to absorb the occasional bandwidth reductions of certain receivers, thereby allowing the FCQUIC source to send faster than if the (occasionally) slow client was directly connected. This, however, increases the implementation complexity and memory consumption of the relays.

4.2.2 Benefits and Trade-offs

From the application perspective, implicit relays require few modifications because neither the source nor the clients need to be aware of them. From the clients' perspective, relays don't add communication round trips and, therefore, do not increase the access time to data.

Relay Migration. One implicit relay limitation is the difficulty in migrating clients to or from relays. For example, if a live-streaming service deployed relays to handle a spike in viewership, it may later find itself with relays serving only a few downstream clients, with no easy way to migrate them without disrupting their service. This limitation is addressed in the explicit relay design (§5).

Client and source authentication. Since FCQUIC implicit relays terminate the session with the source, they have access to the decrypted messages. Clients connecting to a relay must be authenticated to make sure that they are indeed authorized to receive the decrypted contents. Clients may also need to authenticate the relay they are connecting to, this depends on how the relay address was obtained. For example, an address received over a protected connection can be trusted, but one obtained with DNS or from a configuration file may require need to be verified.

Loss of client count. For the Flexicast QUIC source, this approach has a disadvantage in that it no longer has an accurate client count. Historically, one of the reasons for the slow deployment of inter-domain multicast has been difficulty in pricing multicast traffic [3], [62] due to the unpredictability of the bandwidth usage caused by multicast replication. Flexicast QUIC offers a solution by maintaining a unicast path to each client, and therefore always having an accurate client count that can be used for billing. As the protocol stands (i.e., without the addition of fields in frames), implicit relays effectively hide the client count information from the source. This may or may not be an issue depending on the deployment scenario.

4.2.3 Modifications to Flexicast QUIC

A detailed list of the modifications required to the Flexicast QUIC draft [25] to support the implicit relay design is provided in §A.1.

4.3 Implicit Relay implementation

To evaluate the relay design’s ability to reduce an FCQUIC source’s load, we implemented the implicit relay design. The implicit relay is feasible to implement on top of the existing Flexicast QUIC implementation and is sufficient to test and validate (or disprove) the hierarchical relay architecture.

Two implementations of the relay were created, an application level relay, and a version integrated into `tokio-fcquiche`.

4.3.1 Application-level Relay

The application-level relay, hereafter referred to as *Application relay*, is designed to use the existing components exposed by `tokio-fcquiche` to evaluate the potential solution that a `tokio-fcquiche` user may create to alleviate load from their FCQUIC source.

4.3.1.1 Architecture

The relay has two “*sides*”, a client side connected to the upstream FCQUIC source, and a source side distributing data to its downstream clients. The relay must simply “*patch*” the messages from one side to the next, e.g., receiving a message from the receiver side and sending on the source side multicast flow. Each of the sides is a completely separate instance of Flexicast QUIC, no state is shared between them.

Figure 31 presents an overview of the application relay architecture. The upstream source, shown on the left, has its own set of clients. The relay itself, encased in orange, performs the message forwarding explained above. The relay’s downstream clients are shown on the right side.

4.3.1.2 Rust Implementation

`tokio-fcquiche` provides structures and methods to create both of those sides, namely `TokioFcQuicRecv` for the client/receiver and `TokioFcQuic` for the

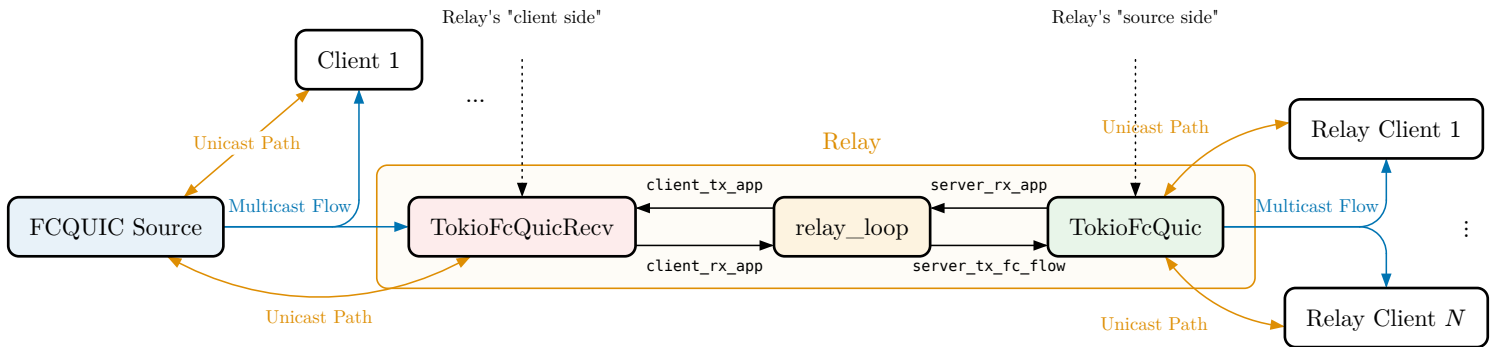


Figure 31: Application relay architecture overview

source. The application relay therefore creates both. Each of those structures exposes communication channels to send and receive messages from the protocol.

The client-side structure `TokioFcQuicRecv` exposes two channels to the application. The first is used by the application to give application data to the protocol, identified by the `Sender` object, named `client_tx_app`. The other channel is used by the protocol to give the application incoming data, notably both from the multicast flow and unicast path in case of retransmissions. This second channel is identified by its `Receiver` side named `client_rx_app`.

The source side is similar, exposing two channels to the application. The first is used by the application to receive data from the protocol (messages sent by the clients on their unicast paths). This channel is identified by the `Receiver` named `server_rx_app`. The second channel is used by the application to transmit data on the multicast flow. It is identified by the `Sender` object named `server_tx_fc_flow`. Each multicast flow has its own `Sender` object, only one is used in this application.

Given the possibly confusing nature of the channels, we can simplify the above description to the following:

1. The relay client side will...

- ... Receive messages from the upstream source's multicast flow OR unicast path on the `client_rx_app` `Receiver` object.
- ... Send messages to the source on its unicast path using the `client_tx_app` `Sender` object.

2. The relay source side will...

- ... Receive messages from its downstream clients (via their unicast paths) on the `server_rx_app` `Receiver` object.
- ... Send messages on its own multicast flow using the `server_tx_fc_flow` `Sender` object.

Two structures are created, one for the relay client-side state (to the source) called `ClientRelayState`, and one for the relay source side state called `ServerRelayState`. The `ClientRelayState` struct establishes the connection to the upstream source. `ServerRelayState` is used by the relay to establish itself as an FCQUIC source. The struct's methods therefore create the `TokioFcQuic` struct as well as the relevant state, and also create a new multicast flow.

These two sides are used in a third structure called `RelayState` that stores the channels described above, as well as the stream IDs to use for communications with each sides. The core of the relay operations take place

in the `relay_loop` method. In an infinite loop, the relay will asynchronously wait on incoming messages from both of the sides at once:

1. **Incoming stream on the client side from the upstream source** (read from `client_rx_app`) will be sent on the multicast flow (via `server_tx_fc_flow`). However, the upstream source assuredly uses different stream IDs for the multicast flow from the relay. The relay must therefore unpack the stream data received from the protocol and send a new stream object using its own multicast flow stream ID value. The relay operates in the same way as the FCQUIC multicast chat server. Each message on the multicast flow is sent using server-initiated unidirectional streams (therefore the stream IDs LSBs are `0x3`).
2. **Incoming stream on the relay's source side from a downstream client** (read from `server_rx_app`) will be forwarded to the upstream source, again using a different stream ID than the one from the client message. In this case, the relay must use client-initiated unidirectional stream IDs (Stream IDs LSBs are `0x2`)

4.3.2 tokio-fcquiche Relay

The `tokio-fcquiche` relay, also referred to as the integrated or FCQUIC relay, uses a similar approach to the application relay for message forwarding, but without the forwarding loop. Message forwarding and stream ID rewriting are handled inside `tokio-fcquiche`, thereby moving the FCQUIC-specific logic into the protocol implementation.

4.3.2.1 Architecture

The integrated relay follows the same design with two sides as the application relay, but does not perform the forwarding in the application. Instead, messages received on one side are directly sent to the other via communication channels. As was the case in the application relay, the integrated relay must also rewrite the stream IDs.

Figure 32 presents this architecture. The diagram shows the upstream FCQUIC source with its own set of clients on the left. The relay maintains a connection to the upstream source via the `TokioFcQuicRecv` structure stored in the `TokioFcQuic` source structure. The channel patching is shown here as an assignment for clarity. The relay's source side maintains the multicast flow as before.

4.3.2.2 Rust Implementation

The `TokioFcQuic` source struct is now capable of acting as a relay. To do so, the relay program enables a flag in the relay's FCQUIC configuration and calls a method to establish a connection to the upstream source. Inter-

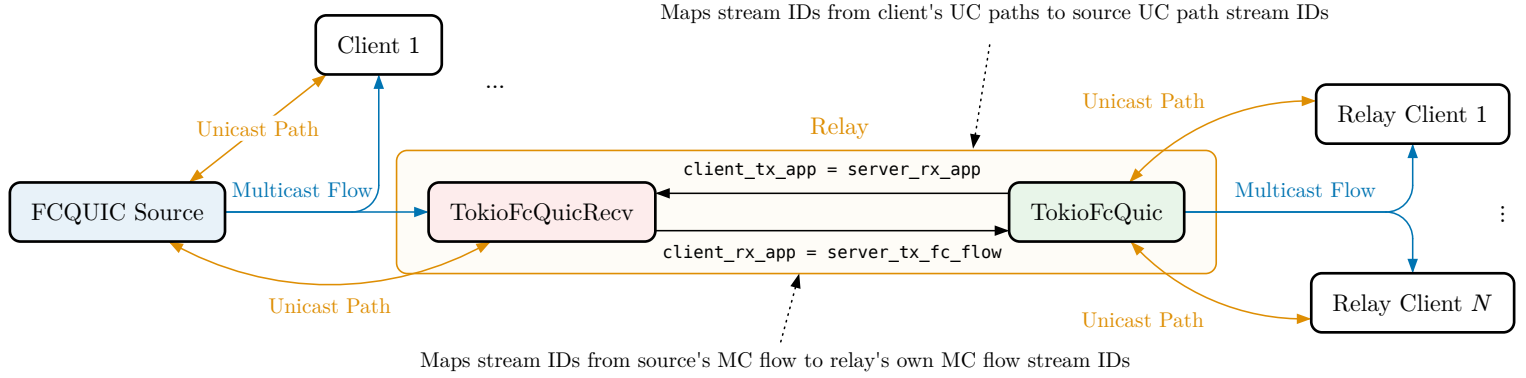


Figure 32: Integrated relay architecture overview

nally, the `TokioFcQuic` struct stores an instance of `TokioFcQuicRecv`. This design still makes use of two `tokio-fcquiche` instances (the source side and the client side), however, the need for an additional relay task has been removed. Instead, the channels exposed by one side are directly passed to the other. For example, the relay's client side connecting to the upstream source exposes two channels (as explained in §4.3.1.2), the receiver end of the channel carrying messages from the source is used by the relay's source side as the receiving end of its application channel. This removes the need for a forwarding loop, as the messages received on one side are directly handed to the other.

Stream ID rewriting is handled by maintaining state on both sides and swapping the stream ID for the appropriate ID before the message is sent on the channel.

4.4 Conclusion of relay design and implementation

In this section, we presented the implicit relay design. Two implementations were created, one using the existing primitives exposed by Flexicast QUIC but requiring knowledge of the protocol's inner workings, and the other integrated inside Flexicast QUIC, allowing for simpler use and potentially improved performance.

4.5 Multi-site evaluation on Grid’5000

To validate the relay design and assess the implementation’s performance, we performed evaluations on Grid’5000. The evaluations compare three deployment scenarios: one with a single FCQUIC source, and two others where the FCQUIC source is supported by an application relay and an integrated relay, respectively.

Grid’5000. The evaluation made extensive use of Grid’5000 resources, a large-scale geographically distributed test-bed dedicated to research in parallel and distributed computing, as well as networking [50]. The test-bed provides access to over 35 clusters spread across 11 sites in France, Belgium, and Luxembourg. Grid’5000 sites are interconnected through RENATER’s 10Gbps backbone.

Grid’5000 is particularly interesting for this work thanks to the bare-metal root access granted to users. Without root access, virtualization would be required, potentially breaking multicast or introducing performance bottlenecks.

4.5.1 Inter-site connectivity

Every Grid’5000 node is assigned a routable IP address (often referred to as its “*production*” address), reachable from any node in any site. Additionally, Grid’5000 allows users to reserve IPv4 /22 subnets (1022 hosts per subnet) on a per site basis, with address assignment within these subnets left to the user. Once assigned, an address from a reserved subnet is reachable from any other site, which we refer to as “globally” routable.

For example, suppose a user reserves 10 nodes and an IPv4 /22 subnet in the *Spiro* cluster in Louvain-la-Neuve. The user can then assign any number of addresses from the reserved subnet to each node, and these addresses will be reachable from any other cluster in Grid’5000.

4.5.2 Multicast support on Grid’5000

The original intent for the relay evaluation was to rely on the network’s support of multicast. However, after carefully testing multicast connectivity between sites, as well as testing the various network configuration tools provided by Grid’5000, it was clear that multicast did not work across sites. It did, however, work between clusters located on the same site (e.g., multicast connectivity works between Lille’s two clusters).

GRE tunneling. To overcome this problem, Generic Routing Encapsulation (GRE, RFC 2784 [17]) tunnels are used between sites. Each cluster can be viewed as a multicast island, connected to other islands using GRE tunnels, as described in section 3.2 of RFC 8313 [16]. In our situation, we

can therefore view a cluster as a domain, with a border router establishing GRE tunnels to other clusters, over which multicast traffic may flow. GRE tunnels must be manually established on both ends of the tunnel. This manual configuration is static and must be established before the start of a multicast stream [16].

Automatic Multicast Tunneling (AMT, RFC 7450 [8]) is another solution described in RFC 8318: it provides automatic multicast delivery from sources in multicast islands to receivers located in other such islands, or in non-multicast networks. This is a more dynamic approach compared to GRE tunnels but requires the introduction of more devices in the forwarding path (AMT gateway and relay) [16]. The GRE tunneling approach was used for its perceived simplicity and for its wide deployment.

Multicast Routers. Every cluster is assigned a *multicast router*. This node is a standalone server that only runs the routing software FRRouting (version 10.4.4) [52]. The routers' production address (preassigned IP address) is used for the GRE tunnel endpoints. OSPF version 2 is the routing protocol used and runs over the GRE tunnels. Each router advertises an IPv4 /24 subnet representing the cluster's hosts. PIM-SSM (Source-Specific-Multicast) is used as the multicast routing protocol. IGMPv3 is used as the multicast group membership protocol.

4.5.3 Experimental setup

The FCQUIC source is placed in a specific cluster, along with a multicast router. Every client cluster contains a multicast router, a relay server, and multiple clients servers.

Network Namespaces. Testing at a larger scale required a larger number of clients. However, running only one client per server would be wasteful of Grid'5000 resources. To increase the number of receivers per client server, Linux network namespaces are used. Connecting network namespaces to the server's production interface requires, among solutions, either to use a bridge, or a MACVLAN [63]. Assigning an IP to a virtual interface that is bridged with others and making this address reachable is not trivial. Instead, MACVLANS allow a physical interface to be bound directly to a namespace without the need for a bridge, giving each network namespace a unique MAC address. However, despite the fact that the network namespaces are connected directly to the production interface, they all share the same broadcast reception queue [64]. This queue contains broadcast and multicast packets, therefore running too many network namespaces on a single server will introduce contention on this queue and lead to sequential access of multicast packets. 5 network namespaces per server is small enough to avoid contention, but large enough to allow testing with 75+ clients.

Address Assignment. To be able to individually address each client, a separate IPv4 /22 subnet was assigned to each cluster. The Flexicast QUIC source, along with the relay nodes, gets assigned one address from the subnet, and each client network namespace is assigned one address (i.e., 5 subnet addresses are assigned to each client node).

Client synchronization. Due to the distributed nature of the experiment, clients must be synchronized to have consistent test durations. To do so, all the clients are given a timestamp for a few seconds in the future. They all connect at their pace and sleep until the timestamp time before starting to send or receive. This avoids issues with late-joining clients sometimes not receiving messages on the multicast flow.

Measure of time. The evaluations are no longer performed on the same machine, therefore, the approach used in the multicast chat no longer applies. Instead, since latencies between sites are a few milliseconds at the very least, the sub millisecond precision of epoch timestamps is sufficient.

Multicast Distribution Tree. The network topology is configurable via a python script. We can define various topologies, including “real” multicast distribution trees. In such configurations, the sites may be connected in series (through tunnels) rather than all connecting directly to the source cluster. For example, a site **A** is directly connected to the source site via a GRE tunnel. Site **B** can establish a GRE tunnel to site **A** and can receive multicast traffic from the source site through site **A**.

4.5.4 Relay Evaluation

In this section, we first describe the evaluated configurations as well as the two tests that were performed. We then look at the results of those tests and draw conclusions.

4.5.4.1 Tests and configurations

Three configurations are evaluated against each other:

1. Clients connect to the FCQUIC source directly.
2. Clients connect to the integrated (FCQUIC) relay located in their cluster.
3. Clients connect to the application relay located in their cluster.

In the latter two configurations, every relay connects directly to the source and joins its multicast flow.

Two tests were designed to evaluate the relay design. First, the **relay overhead test** compares the latency of each configuration when sending equally sized messages. Second, the **data segmentation test** increases the message size to also be larger than a single packet to measure whether the reassembly required at the relays due to segmentation induces more latency than the benefits obtained by offloading work from the source.

4.5.4.2 Topology

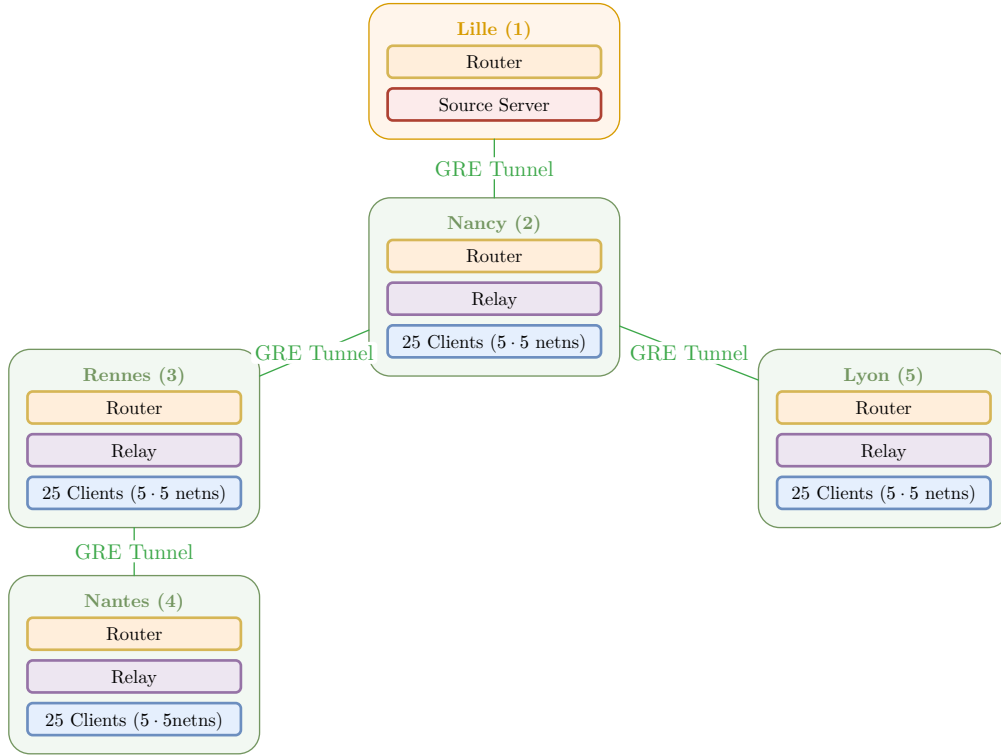


Figure 33: Relay evaluation topology

Figure 33 presents the topology used for the evaluation. A total of 5 clusters were used:

1. Source cluster: *chirop* in Lille (the server node was set to *chirop-5* to avoid machine-to-machine differences)
 - (Servers are *DL360 Gen10+* with an Intel Xeon Platinum 8358 (2 CPUs/node, 32 cores/CPU) and 512 GiB RAM.
2. Client clusters:
 1. *gros* in Nancy (*Dell PowerEdge R640*, Intel Xeon Gold 5220 1 CPU/node, 18 cores/CPU, 96 GiB RAM)
 2. *parasilo* in Rennes (*Dell PowerEdge R630*, Intel Xeon E5-2630 v3, 2 CPUs/node, 8 cores/CPU, 128 GiB RAM)
 3. *ecotype* in Nantes (*Dell PowerEdge R630*, Intel Xeon E5-2630L v4, 2 CPUs/node, 10 cores/CPU, 128 GiB RAM)
 4. *nova* in Lyon (*Dell PowerEdge R430*, Intel Xeon E5-2620 v4, 2 CPUs/node, 8 Cores/CPU, 64 GiB RAM)

Measured one-way latencies between the sites are: Nancy ↔ Lille: 4.3 ms, Rennes ↔ Nancy: 6.0 ms, Nantes ↔ Rennes: 1.1 ms, Lyon ↔ Nancy: 5.6 ms (these values were obtained by halving the RTT measured with ICMP pings between clients in network namespaces, so that the packets are forwarded through the GRE tunnels).

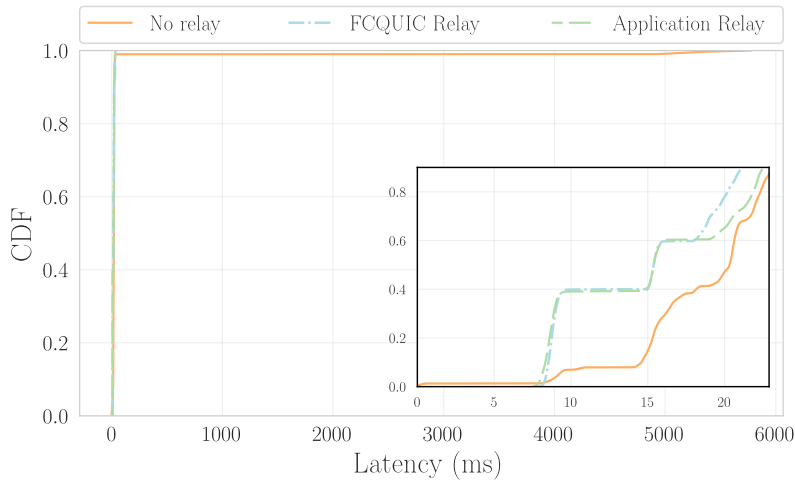
4.5.5 Results

For both tests, no congestion control is used. This is achieved by using an unlimited congestion window. All messages are sent using the same Poisson point process described in §3.2. Each test run lasts for 10 seconds, and is repeated 5 times. Only the CPU usage of the source is monitored and recorded.

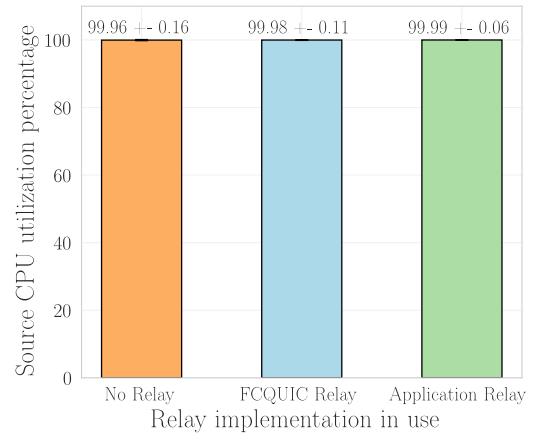
4.5.5.1 Relay overhead

The goal of this first test is to evaluate whether the relay introduces a latency overhead by terminating the source’s multicast tree and distributing its content on another tree. More specifically, we evaluate, when the source is overwhelmed, whether the relay-induced latency is negligible compared to the performance gain obtained by reducing the source’s load. To do so, we deliberately overload the source by saturating it with messages it cannot handle alone.

The sender, located in the `gros` cluster (Nancy) transmits 50 000 messages per second with a 1100-byte payload per message (on top of the timestamp and client ID). The source process is allocated and pinned to 2 threads. This workload was picked to be unmanageable for 2 threads of the source application. The clients, however, are allocated 2 threads each, and the relays 8 threads. Such a high workload was chosen in order to evaluate relays in a scenario that requires them to maintain good service quality.



(a) CDF of latency in milliseconds



(b) Mean CPU utilization of the source for each configuration

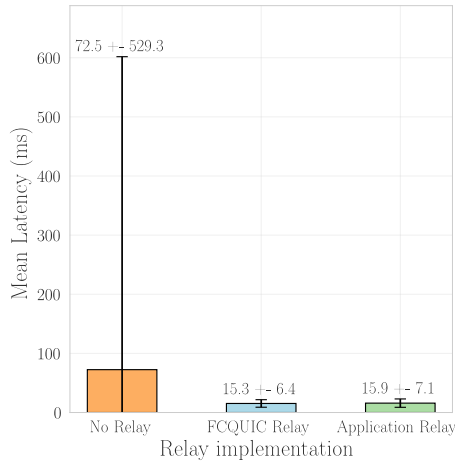
Figure 34: Relay overhead as measured by latency and CPU utilization per configuration

Figure 34a presents the latency CDF per configuration. It is immediately clear, from the shape of the “No Relay” configuration, that the source is not capable of handling the load without relays. The inset window displays the

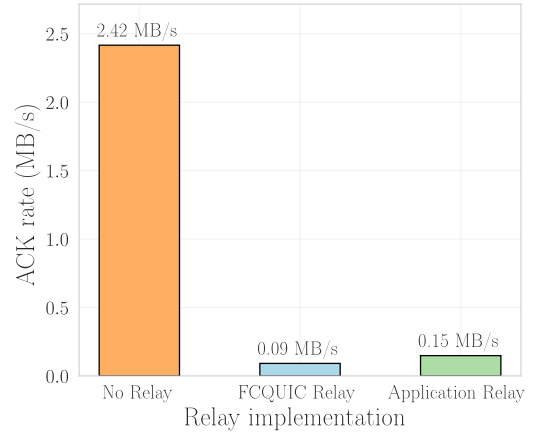
CDF shape until the 90th percentile. Without a relay, the mean latency is 72.4 milliseconds, with a p. 99 latency of 4914 milliseconds. Compared to the configurations where the relay was used shows a large difference: the integrated relay obtains a mean latency of 15.311 ms (4.7 times better than with no relay) and p. 99 latency of 30.470 ms (roughly 161 times better than with no relay). The application-level relay fares similarly, though with a slightly higher latency: with a mean latency of 15.879 ms and p. 99 of 32.406 ms. Figure 35a highlights the mean latency differences between configurations.

Figure 34b presents the source’s mean CPU usage per configuration. Although the latency is no longer as high when relays are used, the source is clearly still under heavy load, as its CPU utilization remains at $\sim 100\%$.

These results seem to imply that, when the source is overloaded and cannot keep up, using relays reduces latency significantly. Additionally, the application relay’s forwarding loop may be responsible for the $\sim 3.7\%$ latency increase compared to the integrated relay.



(a) Mean latency in milliseconds per configuration



(b) Mean ACK rate in MBytes/s

Figure 35: Mean latency and Mean ACK rate of configurations

As previously mentioned, the acknowledgment handling can induce load on the source: this is visible in Figure 35b. Using relays effectively decreases the rate of acknowledgments by up to 16 times, going from 2.42 MB/s, to roughly 0.15 - 0.10 MBytes/s. It must be noted that the value reported for the integrated (FCQUIC) relay is an underestimate, as the QLOG files from which we compute the ACK rate can be truncated at the end of a run, effectively removing a portion of the late ACKs. However, the implementations perform the same ACK suppression role for the source. We therefore expect the true rate to be close, if not identical, to the application relay’s ACK rate

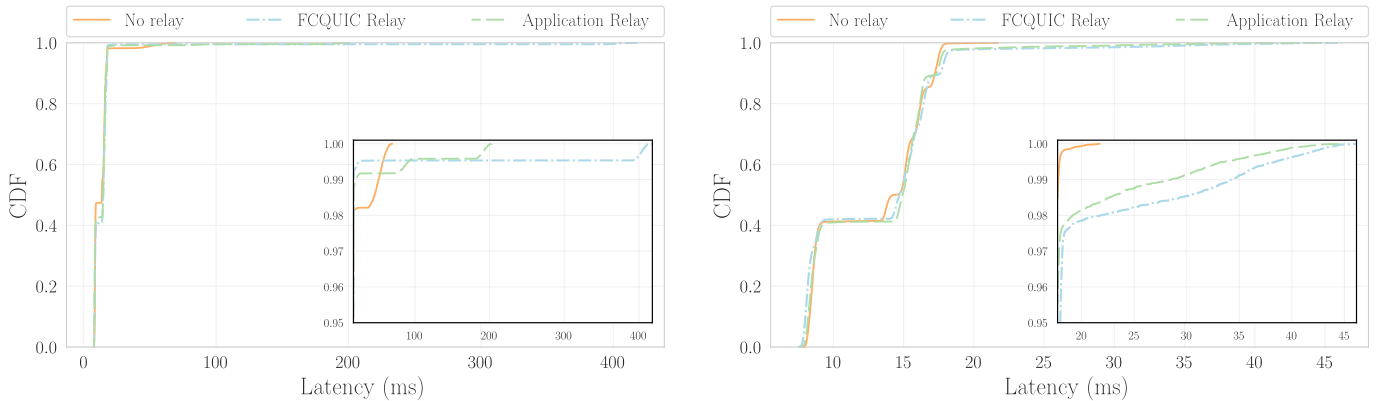
(and vice versa). This value is to be interpreted as a lower bound for the ACK rate.

4.5.5.2 Data segmentation

The goal of this second test is to evaluate the relay implementations with regard to message segmentation. Relays, as currently implemented, must reassemble large messages before transmitting them on their multicast flow. We want to observe whether this reassembly cost outweighs the benefits of offloading work from the source. Contrary to the previous test, the source is not overloaded here, so any latency increase relative to the no-relay baseline can be attributed to relay-induced overhead rather than to the source struggling to keep up.

The source is allocated and pinned to 4 threads. The sender transmission rate is set to 20 000 messages per second. This workload does not overload the source, allowing us to observe the relay’s own induced latency with regard to message reassembly.

The sender, located in the **gros** cluster, starts by sending messages with a 1100-byte payload, then increases to 2200 bytes.



(a) CDF of latency with a 1100-byte payload

(b) CDF of latency with a 2200-byte payload

Figure 36: CDFs of configurations at different message sizes

Figure 36a presents the latency CDF of each configuration when messages carry a 1100-byte payload. At this message size, no segmentation occurs, allowing us to observe the higher mean latency of the relay configurations caused by relay overhead. The no-relay configuration obtains a mean latency of 12.9 ms, while the integrated relay obtains 14.6 ms (a 13% increase) and the application relay obtains 13.6 ms (roughly a 5.5% increase compared to the no-relay).

The relay implementations’ p. 99 latencies are better than the no-relay configuration’s, with the source alone reaching a p. 99 latency of 51.3 ms

compared to 17.9 ms for the integrated relay and 22.6 ms for the application relay, a decrease of 65% and 58% respectively. However, as visible in the long tail of the relays' CDFs, the relays' p. 99 latencies are much higher, reaching 405 ms for the integrated relay and 193.5 ms for the application relay, far exceeding the no-relay configuration's 62.5 ms (do note that this represents very few samples, and may be indicative of a lost packet).

Figure 36b shows the tighter shape of all CDFs. All configurations obtaining similar mean latencies of 12.7ms for no-relay, 13.09 ms for the integrated relay, and 13.08 ms for the application relay. Interestingly, the mean latency of the application relay is yet again slightly lower than the integrated relay. The p. 99 latencies of the relays diverge from the no-relay, with the integrated relay reaching 34.2ms (a 93% increase over no-relay's 17.7 ms p. 99 latency), and the application relay reaching 28.5 ms (60% increase over no-relay). This tail latency increase in the relay implementations may be from segment loss on the source's multicast flow, which would delay message delivery to the relay's downstream clients.

Figure 37a presents the mean latency decrease of the configurations as the message size increases, showing the decrease in the tail and mean latency. Figure 37b confirms the source's capacity in handling this workload, showing its mean CPU utilization remaining stable at $\sim 50\%$.

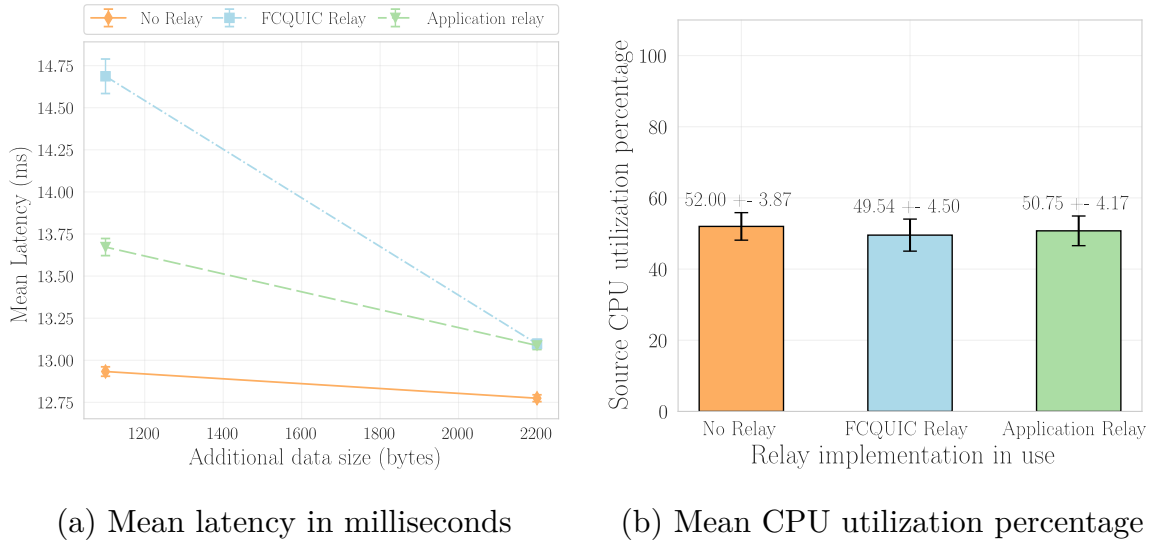


Figure 37: Mean latency and mean CPU utilization of source

From these results, we conclude that the reassembly of messages segmented in two does not significantly increase mean latency, but does tend to increase the tail latency, likely due to segments lost between the source and the relay, forcing the relay to wait for a retransmission before being able to transmit the complete message on its multicast flow.

4.6 Conclusion

This chapter addressed the acknowledgment implosion problem in Flexicast QUIC by proposing a hierarchical architecture, inspired by MoQ relays, that offloads ACK handling and retransmissions from the source. The implicit relay design was presented and implemented in two variants. First, an application-level relay built on top of `tokio-fcquiche` and designed to resemble a solution an end user of Flexicast QUIC could implement. And second, an integrated relay built inside of `tokio-fcquiche`. Another design, called the explicit relay design, is briefly presented and discussed in §5 as a future evolution of this work.

The relay implementations were evaluated on Grid’5000. We created a multi-site distribution tree which spanned over 5 clusters (Lille, Nantes, Nancy, Rennes, and Lyon). GRE tunneling was used to connect each cluster, and FRRouting and PIM-SSM handled multicast routing within the topology. To increase the number of receivers per server, network namespaces were used, with 5 namespaces per node in order to avoid contention when accessing multicast packets from the shared broadcast reception queue.

The relay overhead test overloaded the source by pinning it to 2 threads and using a transmission rate of 50 000 messages per second with a 1100-byte payload. Without a relay, the source could not keep up. The mean latency reached 72.4 ms and the p. 99 latency reached 4914 ms. The integrated relay reduced the mean latency to 15.3 ms (4.7 times lower) and the p. 99 latency to 30.5 ms (roughly 161 times lower). The application relay had similar performance, with a mean latency of 15.9 ms and p. 99 of 32.4 ms, roughly a 3.7% latency increase over the integrated relay likely caused by the forwarding loop of the application relay. The source’s CPU utilization was pinned at $\sim 100\%$ in all configurations (with and without relays), indicating that the source was still very loaded, but the ACK rate dropped from 2.42 MB/s without relays to roughly 0.10 - 0.15 MB/s with relays, a decrease of up to 16 times, indicating a reduction in load for the source from acknowledgement handling.

The data segmentation test decreased the transmission rate, and therefore the workload, to 20 000 messages per second and pinned the server process to 4 threads. This test was designed to measure any latency increase compared to the no-relay version in order to see if it could be attributed to relay overhead rather than to the source struggling. With a 1100-byte payload (implies no message segmentation), the no-relay configuration obtained a mean latency of 12.9 ms, compared to 14.6 ms for the integrated relay (13% increase) and 13.6 ms for the application relay (a 5.5% increase). The p. 99 latencies, however, were much lower for the relays: 17.9 ms for the integrated relay and 22.6 ms for the application relay, compared to 51.3 ms

for the no-relay configuration (decreases of 65% and 58%, respectively). This swapped again after the p. 99 latency, with p. 99.9 latencies of 405 ms for the integrated relay and 193.5 ms for the application relay, both exceeding the no-relay's 62.5 ms. With a 2200-byte payload (segmented in two), the mean latencies somewhat converged: 12.7 ms for no-relay, 13.09 ms for the integrated relay, and 13.08 ms for the application relay. The source's CPU utilization remained at $\sim 50\%$ throughout which confirms that the workload was not overloading the source.

Overall, we conclude that relays can reduce latency when the source is overloaded. This validates the hierarchical architecture as a viable solution to the acknowledgment implosion problem. When the source is not overloaded, however, the relays introduce a small mean latency overhead that seems to disappear as message size grows, but with high p 99.9 tail latencies, likely caused by loss or delay of a segment. Further work could evaluate the use of relays and acknowledgment delays together.

Chapter 5

Future evolution: Explicit Relay Design

The implicit relay design presented and evaluated in the previous chapter is transparent both to the FCQUIC source and all clients. This reduces the required protocol changes and therefore does not require many modifications to the existing Flexicast QUIC implementation. However, it has several limitations: the FCQUIC source loses the accurate client count, clients and relays must now authenticate each other outside of Flexicast QUIC (out of band, e.g., through the application), and client migration between relays (or back to the source) is not trivial and may require disrupting the client’s service.

These limitations are due to the transparent nature of this first design, therefore, to avoid these issues the relay’s presence and role must be made explicit at the protocol level to the source and the clients. In this chapter, we describe a second relay design, the *explicit relay*, that makes the relay known to others, but also defines the changes required to alleviate the previously detailed issues. This design requires modifications to the Flexicast QUIC protocol and implementation, therefore it must be seen as a future evolution of the work in this thesis. A description of the modifications required to the Flexicast QUIC draft [25] to add explicit relays is provided in annex §A.2.

5.1 Design

The explicit relay makes the relays known to both the source and the clients. During the handshake with a source, a relay announces its role via a new transport parameter or a new FC_STATE action. The FCQUIC source maintains a list of active relays and can redirect new or existing clients to a relay. As before, the relay joins the multicast flow as a client and redistributes its content on a new multicast tree.

Client connection. The client connection process may potentially be longer. During the initial connection to the FCQUIC source, the client learns of the source’s multicast flow using an FC_ANNOUNCE frame, but it may now also learn of the presence of relays using the same FC_ANNOUNCE frame, or with a new FC_RELAY_INFO frame. The source has the option to either advertise all known relays, or only advertise specific ones (e.g., nearby or less loaded relays). Upon learning of a relay, the clients may choose to disconnect from the source and connect to it instead. The relay then informs the source of its updated client count or client list using a new frame.

Message caching. Explicit relays support message caching in a similar fashion to implicit relays. Since the source is aware of the relays, it has finer control over the caching behavior. The source can disallow caching with a new transport parameter exchanged during the QUIC handshake. When caching is disabled, the retransmission latency is higher, as the source must handle retransmissions directly. When enabled, however, the relay will retain messages and handle retransmissions locally. A new frame (e.g., `RELAY_AGG_ACK`) may be introduced to let the relay send acknowledgment information and network condition information to the source. This frame can also carry other information, such as the relay’s client count and buffer occupancy, to inform the source of increasing buffering. Without this new frame, the source may be disconnected from the network conditions happening beyond the relays, letting relays deal with potential loss and congestion while the source continues to transmit as fast as the relay’s network will allow.

5.2 Congestion Control with relays

Flexicast QUIC supports a source distributing data on multiple multicast flows. Each flow has its own set of keys, its own bitrate, and can even carry different information. A Flexicast QUIC source may use multiple multicast flows to send data at varying bitrates, or at a constant bitrate, but shifted in time.

A relay, once again, has two options: 1) Join one multicast flow, buffer data, and expose multiple multicast flows for its downstream clients. Or 2) Join multiple multicast flows (as required), acknowledge messages from the flows at the exact rate required for each one, and redistribute the flow information on new multicast flows.

In single multicast flow operations, the source adapts the flow bitrate to an acceptable level for all receivers. A slow receiver will lead to a bitrate reduction, if it remains too slow, the source eventually removes it from the multicast flow [25]. The same approach can be applied to relays. The relay may acknowledge the source’s messages at a specific bitrate, dynamically adjusting it based on its downstream clients’ bitrate. Alternatively, it may aggregate the `MAX_DATA` and `MAX_STREAM_DATA` frames received from its clients and forward them to the source. `MAX_DATA` and `MAX_STREAM_DATA` are QUIC frames that contain the maximum amount of data that can be sent on a connection and on a stream, respectively [19].

5.3 Relay migration

One of the trade-offs highlighted for the implicit relay design was the difficulty of migrating clients between relays, or from a relay back to the source. In the explicit relay model, we are free to define new frames to simplify

this scenario. Consequently, a new **LEAVING** frame may be sent by the relay to the source to announce its intention to stop operating. In response, the source may provide the relay with a new relay address to distribute to the downstream clients. Upon reception of this new address, the clients may directly connect to the new relay and join its multicast flow.

5.4 Benefits and Trade-offs

The main advantage of explicit relays is that they remove the need for out-of-band communication, along with client and relay authentication. In this design, the authentication is no longer necessary because the client receives the relay's information directly from the source. Additionally, the source maintains full information about the multicast distribution tree, such as client count and network conditions.

Nevertheless, the client connection will take more time to establish, as it must first connect to the source, learn the existence of a relay, connect to the relay, and join its multicast flow. The source's implementation is also required to maintain more state and handle more messages on the reverse path.

5.5 Conclusion

The explicit relay design mitigates some of the main limitations of the implicit relay design but it requires more protocol changes as well as a longer client connection process. Implementing this design was outside the scope of this work. The lessons learned from the implicit relay design and implementation seem to make a case for this explicit relay design. Specifically, the difficult client migration, loss of an accurate client count, and need for out-of-band authentication of relays and clients. It remains to be seen, however, if such an implementation would provide enough gains over the implicit relay to justify the modifications.

Chapter 6

Conclusion

This master’s thesis focused on Flexicast QUIC’s ability to efficiently and reliably deliver application data for less bandwidth intensive applications. We started by introducing the relevant background and discussing existing solutions.

6.1 Multicast Chat

First, in chapter 3 we defined and implemented a multicast chat application using Flexicast QUIC’s `tokio-fcquiche` library and built baselines using Cloudflare’s QUIC library `tokio-quiche` and TCP (with TLS). These implementations were evaluated on simulated topologies in network namespaces. The evaluation allowed us to measure Flexicast QUIC’s implementation overhead on very small topologies, showing that for very small use cases a unicast-based approach uses significantly less CPU and provides lower latency. Adding network loss while maintaining a small number of receivers allowed us to observe QUIC’s head-of-line blocking avoidance, which contributed to Flexicast QUIC retaining a low p. 99 latency. Increasing the network size to 28 receivers let us measure the multicast-induced latency advantage of Flexicast QUIC. Its mean latency was 5.49 ms, on par with the baseline QUIC version and the non-TLS TCP version, and lower than TCP+TLS’s 7.10 ms. The tail latency was also in favor of FCQUIC, with an 11 times reduction compared to TCP+TLS.

Increasing the number of receivers further, up to 81 in steps of 10, clearly showed the median latency increase of all unicast-based approaches going from 4 ms to 4.3 ms, while FCQUIC’s latency hovered at 4.1 ms. It also highlighted the large increase in tail latency, with TCP+TLS’s p. 99 latency reaching up to 27 times that of Flexicast QUIC.

The payload sensitivity test evaluated the impact of large message sizes on the implementations. We observed a latency increase from repeated encryption operations. Although this increase was present in all implementations, it affected TCP+TLS more than the others, with a tail latency 23 times higher than FCQUIC and the other baselines. FCQUIC’s median latency followed the same trend as the baselines, albeit more steadily, and remained slightly lower by about 0.1 ms (4.5 ms median latency at 9000 payload bytes for FCQUIC vs 4.65 ms for the baselines).

However, throughout these tests, the CPU utilization of the Flexicast QUIC implementation remained consistently high, hovering at 40 to 50%

under no to light load and increasing to ~60% under load, while the baselines all hovered at 0 to 1% under light load and up to ~5% under load.

Overall, FCQUIC’s implementation overhead is measurable on small topologies and is reflected in consistently high CPU utilization across all tests. This overhead has a fixed cost at low speeds such as the ones used for this evaluation. This cost, however, is quickly absorbed by the latency gains, as the number of receivers, message size, or network loss increases.

6.2 Flexicast QUIC Relays

Implicit Relay Design. Second, in chapter 4 we moved away from comparing FCQUIC to baselines and instead placed FCQUIC in a scenario where a single source serves many receivers and cannot cope with the volume of acknowledgements. To address this *acknowledgment implosion* problem, we introduced a hierarchical architecture to offload ACK handling and retransmissions from the source, with *relay nodes* placed in the tree to act as intermediate sources. The implicit relay design was presented, with the main disadvantage of hiding information from the source and clients and difficult client migration, but ease of implementation. Two implementations of the design were created: one using the primitives exposed by Flexicast QUIC to evaluate how well an FCQUIC end user could solve this problem, and one integrated directly inside `tokio-fcquiche`.

Both implementations were evaluated on Grid’5000 to form a large-scale, multi-site multicast distribution tree spanning 5 clusters, with the source in Lille and receivers distributed across France, the farthest being in Lyon. The relays successfully helped an overloaded source cope with the traffic, reducing the mean latency by 4.7 times and the p. 99 tail latency by as much as 160 times. The latency overhead of placing relays on the forwarding path was also evaluated, showing a small increase of 5 to 13% when messages are small enough to fit within a single QUIC packet, and essentially disappearing as the message size increases.

We find that relays are an effective solution to reduce the high tail latency induced by an overloaded source. The integrated relay implementation, however, showed little latency benefit over the application-level variant, validating that an end user can implement their own relay design and successfully offload work from the source. In cases where the source was not overloaded, relays introduced only a small latency overhead.

Explicit Relay Design. Finally, in chapter 5 we discussed another relay design, the explicit relay, whose role and presence is no longer hidden from the source and clients, which addresses several limitations of the implicit relay design. However, the design requires a lot of protocol and implementation changes and would also result in longer client connection establishment. This

design was not implemented but was presented as a possible extension of this work.

6.3 Source code

The source code for this work can be found in the following Github repository: https://github.com/zeykafx/fcquic_applications_master_thesis

The repository contains several submodules:

- **Flexicast QUIC fork with relay code:** <https://github.com/zeykafx/flexicast-quic>
- **Multicast chat application:** https://github.com/zeykafx/fcquic_chat
- **Flexicast QUIC relay implementations:** https://github.com/zeykafx/fcquic_relay

Generative AI Disclosure

Generative AI (LLMs or otherwise) were used in the following ways during this work:

1. DeepL Write and Claude Sonet 4.6 were used to correct spelling issues and fix grammar issues. No text used in this document has been generated by an LLM from scratch and used as is, i.e. all the text has been written by me based on content from the cited sources, or based on personal notes, code, etc. Any grammar changes suggested by an LLM were reviewed, sometimes modified or rejected. The majority of the text has not been passed through LLMs.
2. During development, LLMs were used in the following ways:
 - Github copilot's edit prediction was not used.
 - Gemini 3.1 Pro and Claude Opus were very occasionally called via Github copilot to find tricky bugs (though it ultimately didn't prove very useful), and occasionally to fix issues in graph code.
 - Gemini was used to merge together a series of bash scripts used to run each specific chat app implementation (e.g., `run_tcp_server.sh`, `run_baseline_server.sh`) into a python script. This python file was then reviewed and modified, but given its non critical nature, the code remains largely LLM generated code based on my bash scripts.
 - The most significant use of an LLM was for the relay evaluation test script. Given the large number of servers to coordinate (4 client clusters with 25 clients each, 1 relay server per cluster, and 1 cluster with the source server), NPF did not always manage to start the required processes at the right time or reliably capture results. For a short test run of 5 to 10 seconds, NPF would very often take up to 15 minutes to complete, and would, in most cases, capture few if any results. After 4 to 5 days of work attempting to debug the script, I asked Gemini 3.1 Pro to rewrite the NPF script as a Python script that performed the same task. The resulting script was close to the requirements, and after a thorough review and many modifications, the script (located in the relay evaluation notebook) performs well and is able to orchestrate the server, clients, and relays. The original code structure remains AI generated, but extensive modifications were made and many parts were rewritten to do the exact task I need.

Bibliography

- [1] B. Rajagopalan, “Reliability and scaling issues in multicast communication,” in *Conference proceedings on Communications architectures & protocols*, in SIGCOMM '92. New York, NY, USA: Association for Computing Machinery, Oct. 1992, pp. 188–198. doi: [10.1145/144179.144275](https://doi.org/10.1145/144179.144275).
- [2] S. E. Deering and D. R. Cheriton, “Multicast routing in datagram internetworks and extended LANs,” *ACM Transactions on Computer Systems (TOCS)*, vol. 8, no. 2, pp. 85–110, May 1990, doi: [10.1145/78952.78953](https://doi.org/10.1145/78952.78953).
- [3] C. Diot, B. Levine, B. Lyles, H. Kassem, and D. Balensiefen, “Deployment issues for the IP multicast service and architecture,” *IEEE Network*, vol. 14, no. 1, pp. 78–88, Jan. 2000, doi: [10.1109/65.819174](https://doi.org/10.1109/65.819174).
- [4] L. Navarre and O. Bonaventure, “Towards an Internet Deployment of Flexible Multicast QUIC,” in *Proceedings of the ACM SIGCOMM 2025 Posters and Demos*, Coimbra Portugal: ACM, Sept. 2025, pp. 137–139. doi: [10.1145/3744969.3748403](https://doi.org/10.1145/3744969.3748403).
- [5] J. Maisonneuve *et al.*, “An Overview of IPTV Standards Development,” *IEEE Transactions on Broadcasting*, vol. 55, no. 2, pp. 315–328, June 2009, doi: [10.1109/TBC.2009.2020451](https://doi.org/10.1109/TBC.2009.2020451).
- [6] secmaVenger, “Use multicast to deploy Windows over the network - Configuration Manager.” Accessed: May 25, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/intune/configmgr/osd/deploy-use/use-multicast-to-deploy-windows-over-the-network>
- [7] T. DeMeyer, “Netflix glitches plague popular live stream of Jake Paul-Mike Tyson card,” *The New York Times*, Jan. 2025, Accessed: Nov. 20, 2025. [Online]. Available: <https://www.nytimes.com/athletic/5925844/2024/11/16/netflix-buffering-freezing-jake-paul-mike-tyson/>
- [8] G. Bumgardner, “Automatic Multicast Tunneling,” Request for Comments RFC7450, Feb. 2015. doi: [10.17487/RFC7450](https://doi.org/10.17487/RFC7450).
- [9] J. Holland, “DNS Reverse IP Automatic Multicast Tunneling (AMT) Discovery,” Request for Comments RFC8777, Apr. 2020. doi: [10.17487/RFC8777](https://doi.org/10.17487/RFC8777).
- [10] J. Holland, L. Pardue, M. Franke, and K. Rose, “Multicast Extension for QUIC,” Internet Draft draft-jholland-quic-multicast-08, Jan. 2026. Accessed: May 25, 2026. [Online]. Available: <https://datatracker.ietf.org/doc/draft-jholland-quic-multicast-08>
- [11] L. Navarre, Q. De Coninck, T. Barbette, and O. Bonaventure, “Taking the Best of Multicast and Unicast with Flexicast QUIC,” *SIGCOMM Comput. Commun. Rev.*, vol. 55, no. 2, pp. 2–12, July 2025, doi: [10.1145/3750832.3750834](https://doi.org/10.1145/3750832.3750834).
- [12] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei, “The PIM architecture for wide-area multicast routing,” *IEEE/ACM Transactions on Networking*, vol. 4, no. 2, pp. 153–162, Apr. 1996, doi: [10.1109/90.490743](https://doi.org/10.1109/90.490743).
- [13] H. Holbrook and B. Cain, “Source-Specific Multicast for IP,” Request for Comments RFC4607, Aug. 2006. doi: [10.17487/RFC4607](https://doi.org/10.17487/RFC4607).

- [14] B. Haberman, “Internet Group Management Protocol, Version 3,” Request for Comments RFC9776, Mar. 2025. doi: [10.17487/RFC9776](https://doi.org/10.17487/RFC9776).
- [15] S. Bhattacharya, “An Overview of Source-Specific Multicast (SSM),” Request for Comments RFC3569, July 2003. doi: [10.17487/RFC3569](https://doi.org/10.17487/RFC3569).
- [16] P. Tarapore, R. Sayko, G. Shepherd, T. Eckert, and R. Krishnan, “Use of Multicast across Inter-domain Peering Points,” Request for Comments RFC8313, Jan. 2018. doi: [10.17487/RFC8313](https://doi.org/10.17487/RFC8313).
- [17] T. Li, D. Farinacci, S. P. Hanks, D. Meyer, and P. S. Traina, “Generic Routing Encapsulation (GRE),” Request for Comments RFC2784, Mar. 2000. doi: [10.17487/RFC2784](https://doi.org/10.17487/RFC2784).
- [18] L. Giuliano, C. Lenart, and R. Adam, “TreeDN: Tree-Based Content Delivery Network (CDN) for Live Streaming to Mass Audiences,” Request for Comments RFC9706, Jan. 2025. doi: [10.17487/RFC9706](https://doi.org/10.17487/RFC9706).
- [19] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Request for Comments RFC9000, May 2021. doi: [10.17487/RFC9000](https://doi.org/10.17487/RFC9000).
- [20] A. Langley *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*, in SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, Aug. 2017, pp. 183–196. doi: [10.1145/3098822.3098842](https://doi.org/10.1145/3098822.3098842).
- [21] Q. De Coninck and O. Bonaventure, “Multiflow QUIC: A Generic Multipath Transport Protocol,” *IEEE Communications Magazine*, vol. 59, no. 5, pp. 108–113, May 2021, doi: [10.1109/MCOM.001.2000892](https://doi.org/10.1109/MCOM.001.2000892).
- [22] 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-21, Mar. 2026. Accessed: May 22, 2026. [Online]. Available: <https://datatracker.ietf.org/doc/draft-ietf-quic-multipath>
- [23] Q. De Coninck and O. Bonaventure, “Multipath QUIC: Design and Evaluation,” in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, in CoNEXT '17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 160–166. doi: [10.1145/3143361.3143370](https://doi.org/10.1145/3143361.3143370).
- [24] Q. De Coninck, “The packet number space debate in multipath QUIC,” *ACM SIGCOMM Computer Communication Review*, vol. 52, no. 3, pp. 2–9, Sept. 2022, doi: [10.1145/3561954.3561956](https://doi.org/10.1145/3561954.3561956).
- [25] L. Navarre and O. Bonaventure, “Flexicast QUIC: combining unicast and multicast in a single QUIC connection,” Internet Draft draft-navarre-quic-flexicast-02, Feb. 2026. [Online]. Available: <https://datatracker.ietf.org/doc/draft-navarre-quic-flexicast>
- [26] L. Navarre, “Can you Multicast? Deploying Flexible Multicast on the internet with Flexicast QUIC,” *SIGCOMM*, 2026.

- [27] D. Duplyakin *et al.*, “The design and operation of cloudlab,” in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, in USENIX ATC '19. USA: USENIX Association, July 2019, pp. 1–14.
- [28] “cloudflare/quiche: Savoury implementation of the QUIC transport protocol and HTTP/3.” Accessed: Nov. 20, 2025. [Online]. Available: <https://github.com/cloudflare/quiche>
- [29] “IPNetworkingLab/flexicast-quic.” Accessed: May 18, 2026. [Online]. Available: <https://github.com/IPNetworkingLab/flexicast-quic>
- [30] “tokio-rs/tokio.” Accessed: May 11, 2026. [Online]. Available: <https://github.com/tokio-rs/tokio>
- [31] “Instant in std::time - Rust.” Accessed: May 18, 2026. [Online]. Available: <https://doc.rust-lang.org/std/time/struct.Instant.html>
- [32] J. Cao, W. S. Cleveland, D. Lin, and D. X. Sun, “Internet Traffic Tends Toward Poisson and Independent as the Load Increases,” *Nonlinear Estimation and Classification*. Springer, New York, NY, pp. 83–109, 2003. doi: 10.1007/978-0-387-21579-2_6.
- [33] V. Paxson and S. Floyd, “Wide area traffic: the failure of Poisson modeling,” *IEEE/ACM Transactions on Networking*, vol. 3, no. 3, pp. 226–244, June 1995, doi: 10.1109/90.392383.
- [34] S. Floyd and V. Paxson, “Difficulties in simulating the internet,” *IEEE/ACM Transactions on Networking (TON)*, vol. 9, no. 4, pp. 392–403, Aug. 2001, doi: 10.1109/90.944338.
- [35] “rkyv/rkyv.” Accessed: Apr. 18, 2026. [Online]. Available: <https://github.com/rkyv/rkyv>
- [36] T. Barbette, “Poster: NPF: orchestrate and reproduce network experiments.”
- [37] “quiche/tokio-quiche at master · cloudflare/quiche.” Accessed: May 08, 2026. [Online]. Available: <https://github.com/cloudflare/quiche/tree/master/tokio-quiche>
- [38] “Async QUIC and HTTP/3 made easy- tokio-quiche is now open-source.” Accessed: May 18, 2026. [Online]. Available: <https://blog.cloudflare.com/async-quic-and-http-3-made-easy-tokio-quiche-is-now-open-source/>
- [39] L. Navarre, “flexicast-quic/apps/src/bin/fc-server-rtp-tokio.rs at sigcomm-ccr · IPNetworkingLab/flexicast-quic.” Accessed: May 08, 2026. [Online]. Available: <https://github.com/IPNetworkingLab/flexicast-quic/blob/sigcomm-ccr/apps/src/bin/fc-server-rtp-tokio.rs>
- [40] “flexicast-quic/apps/src/mc_app/asynchronous/uc.rs at sigcomm-ccr · IPNetworkingLab/flexicast-quic.” Accessed: May 08, 2026. [Online]. Available: https://github.com/IPNetworkingLab/flexicast-quic/blob/sigcomm-ccr/apps/src/mc_app/asynchronous/uc.rs
- [41] “flexicast-quic/apps/src/mc_app/asynchronous/controller.rs at sigcomm-ccr · IPNetworkingLab/flexicast-quic.” Accessed: May 08, 2026. [Online]. Available: https://github.com/IPNetworkingLab/flexicast-quic/blob/sigcomm-ccr/apps/src/mc_app/asynchronous/controller.rs

- [42] “Examining HTTP/3 usage one year on.” Accessed: May 18, 2026. [Online]. Available: <https://blog.cloudflare.com/http3-usage-one-year-on/>
- [43] “Congestion Control in IP/TCP Internetworks,” Request for Comments RFC896, Jan. 1984. doi: [10.17487/RFC0896](https://doi.org/10.17487/RFC0896).
- [44] R. T. Braden, “Requirements for Internet Hosts - Communication Layers,” Request for Comments RFC1122, Oct. 1989. doi: [10.17487/RFC1122](https://doi.org/10.17487/RFC1122).
- [45] “tcp.h « net « include - kernel/git/torvalds/linux.git - Linux kernel source tree.” Accessed: May 11, 2026. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/net/tcp.h?h=v7.1-rc3#n156>
- [46] “tcp(7) - Linux manual page.” Accessed: May 10, 2026. [Online]. Available: <https://man7.org/linux/man-pages/man7/tcp.7.html>
- [47] J. Padhye, S. Floyd, and M. J. Handley, “TCP Congestion Window Validation,” Request for Comments RFC2861, June 2000. doi: [10.17487/RFC2861](https://doi.org/10.17487/RFC2861).
- [48] G. Fairhurst, A. Sathiaselalan, and R. Secchi, “Updating TCP to Support Rate-Limited Traffic,” Request for Comments RFC7661, Oct. 2015. doi: [10.17487/RFC7661](https://doi.org/10.17487/RFC7661).
- [49] “tcp.h « net « include - kernel/git/torvalds/linux.git - Linux kernel source tree.” Accessed: May 11, 2026. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/net/tcp.h?h=v7.1-rc3#n164>
- [50] D. Balouek *et al.*, “Adding Virtualization Capabilities to the Grid’5000 Testbed,” in *Cloud Computing and Services Science*, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds., Cham: Springer International Publishing, 2013, pp. 3–20. doi: [10.1007/978-3-319-04519-1_1](https://doi.org/10.1007/978-3-319-04519-1_1).
- [51] “Getting Started - Grid5000.” Accessed: May 12, 2026. [Online]. Available: https://www.grid5000.fr/w/Getting_Started#Storage
- [52] “FRRouting.” [Online]. Available: <https://frrouting.org/>
- [53] “Use of OSI IS-IS for routing in TCP/IP and dual environments,” Request for Comments RFC1195, Dec. 1990. doi: [10.17487/RFC1195](https://doi.org/10.17487/RFC1195).
- [54] A. Doeraene, “Design of a multicast congestion control algorithm for Multicast QUIC,” 2025. Accessed: May 11, 2026. [Online]. Available: <https://hdl.handle.net/2078.2/43780>
- [55] “FFSexp3-master-thesis/evaluation at main · Aperature/FFSexp3-master-thesis.” Accessed: May 10, 2026. [Online]. Available: <https://github.com/Aperature/FFSexp3-master-thesis/tree/main/evaluation>
- [56] R. Dodin, “containerlab.” Accessed: May 11, 2026. [Online]. Available: <https://containerlab.dev/>
- [57] “Measuring Fixed Broadband - Thirteenth Report | Federal Communications Commission.” Accessed: May 20, 2026. [Online]. Available: <https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-thirteenth-report>

- [58] R. Richter, V. Ververis, and V. Bajpai, “Breaking Through the Clouds: Performance Insights into Starlink’s Latency and Packet Loss,” presented at the IFIP Netw. Conf., Limassol, Cyprus, May 2025.
- [59] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya, “Reliable Multicast Transport Protocol (RMTP),” *Selected Areas in Communications, IEEE Journal on*, vol. 15, pp. 407–421, May 1997, doi: [10.1109/49.564138](https://doi.org/10.1109/49.564138).
- [60] J. P. Macker, C. Bormann, M. J. Handley, and B. Adamson, “NACK-Oriented Reliable Multicast (NORM) Transport Protocol,” Request for Comments RFC5740, Nov. 2009. doi: [10.17487/RFC5740](https://doi.org/10.17487/RFC5740).
- [61] S. Nandakumar, V. Vasiliev, I. Swett, and A. Frindell, “Media over QUIC Transport,” Internet Draft draft-ietf-moq-transport-18, May 2026. Accessed: May 16, 2026. [Online]. Available: <https://datatracker.ietf.org/doc/draft-ietf-moq-transport>
- [62] T. N. H. Henderson and S. N. Bhatti, “Protocol-independent multicast pricing.”
- [63] H. Liu, “Introduction to Linux interfaces for virtual networking.” Accessed: May 19, 2026. [Online]. Available: <https://developers.redhat.com/articles/2026/04/03/introduction-to-linux-interfaces-for-virtual-networking>
- [64] “ip-link(8) - Linux manual page.” Accessed: May 19, 2026. [Online]. Available: <https://www.man7.org/linux/man-pages/man8/ip-link.8.html>
- [65] C. F. Jennings, S. Nandakumar, and R. Barnes, “End-to-End Secure Objects for Media over QUIC Transport,” Internet Draft draft-ietf-moq-secure-objects-00, Mar. 2026. Accessed: May 16, 2026. [Online]. Available: <https://datatracker.ietf.org/doc/draft-ietf-moq-secure-objects-00>
- [66] “tc-netem(8) - Linux manual page.” Accessed: May 11, 2026. [Online]. Available: <https://man7.org/linux/man-pages/man8/tc-netem.8.html>

A Appendix

A.1 Modifications to Flexicast QUIC for the Implicit Relay

Given the transparent nature of this relay, most sections of the Flexicast QUIC draft [25] do not require modifications. The relay manages its own multicast flow, which allows it to reuse the existing implementation on both sides of the relay. Only three sections are affected.

Section 5.1 (*Flexicast Flow Announcement*) should be updated to describe how the relay can advertise its own multicast flow using an FC_ANNOUNCE frame.

Section 7 (*Reliability*) should be updated to take into account how the relay acknowledges messages. If the relay caches messages, it acknowledges them directly to the source, making it appear as a fast client for the source. If it does not cache messages, it should aggregate acknowledgments from its clients and send an aggregated acknowledgment to the source, letting it handle potential retransmissions.

Section 12 (*Discussion*) should discuss the implicit relay model as a potential deployment strategy. It should also discuss the authentication problems: 1) how to prevent unauthorized clients from receiving content, and 2) how to protect clients from connecting to unauthorized/illegitimate relays. Potential solutions include pushing authentication to the application layer, or using two layer encryption (similarly to the two-layer encryption approach described in Media-over-QUIC's end-to-end secure objects draft [65]), where the FCQUIC source encrypts messages with a key shared out of band by the application and lets TLS in QUIC encrypt the message. In this end-to-end encryption scheme, a relay can decrypt the QUIC packet but cannot access the encrypted message payload.

A.2 Modifications to Flexicast QUIC for the Explicit Relay

The explicit relay design requires more modifications to the Flexicast QUIC draft [25] than the implicit relay, due to the fact that the source must be aware of the relays and the protocol must support the interactions between them as detailed previously.

Section 4 (*Handshake Negotiation and Transport parameter*) should be updated to describe how a relay announces its role to the source during, or after the handshake. This may be performed with a new transport parameter (e.g., flexicast_relay_support), or with a new FC_STATE action.

Section 5 (*Initialization of a Flexicast Flow*) should be extended to describe the relay connection flow. Figure 2 (visual overview of the multicast announcement and join by a receiver) should be updated in order to illustrate how clients may learn of the relay from the source, then may connect to it and join its multicast flow. To distribute the relay information, the source may either: send an extended FC_ANNOUNCE frame that now carries relay information (e.g., addresses of all relays, or a chosen relay for the client), or send a new FC_RELAY_INFO frame that carries this information separately.

Section 5.1 (*Flexicast Flow Announcement*) should make clear that the FC_ANNOUNCE frame may carry either the source's or the relay's address. It should also handle the case where a relay is leaving, and the clients must be sent information about a new relay. FC_ANNOUNCE may be used to inform the clients of a new relay address.

Section 5.2 (*Joining a Flexicast Flow*) the receiver-side finite-state machine should be updated to account for the relay path, starting with the client being unaware of the relay, to learning of its existence from the source, to then connecting to the relay, and finally to joining its multicast flow.

Section 6 (*Multicast flow membership management*) should be updated to mention that the source can track the directly connected clients as well as downstream clients of relays.

Section 6.1 (*Receiver-side management*) should be updated to include the procedure for the case when a relay must disconnect/leave. The relay starts by sending a new LEAVING frame to the source, the source responds with an updated relay list or a single chosen relay address, and then the relay forwards this information to its clients.

Section 7 (*Reliability*) should include a new subsection on the aggregation of acknowledgments. 1) In non-caching mode, the relay forwards messages from the source on its multicast flow, aggregates acknowledgments from its downstream clients, and lets the source retransmit lost messages on the relay's unicast path. The relay forwards these retransmissions to the affected client on their unicast path. This non-caching mode is simple to implement but incurs high retransmission latency. 2) In caching mode, the relay stores messages until all clients acknowledge them, handles retransmissions locally, and sends detailed acknowledgment information to the source via a new frame such as RELAY_AGG_ACK. This new frame informs the source of the network condition.

Section 8 (*Congestion Control*) should describe that a relay can join multiple multicast flows to redistribute them to its downstream clients as well as how it matches the source's flow bitrates. The section should also describe the potential approaches for constant bitrates: 1) the relay may set its PATH_ACK rate to the bitrate of its slowest client (essentially acknowledging

messages from the source at the pace of its slowest client, a bit of a hack), 2) or it may send bitrate information to the source via a new frame such as `RELAY_CONGESTION_INFO`.

Section 9 (*Flow Control*) should describe the relay aggregation of the `MAX_DATA` and `MAX_STREAM_DATA` frames from the downstream clients and forwarding of the minimum values to the source, possibly in the same frames.

Section 11.2.1 (*FC_STATE actions*) should introduce a new action for the relay. 1) A `RELAY` action may be used to let the relay announce its role to the source, and a `RELAY_LEAVING` action may be used by the relay to inform the source of its upcoming disconnection.

A.3 Topology script configuration

The topology scripts accept a `YAML` configuration file. Each router and link is individually configurable, allowing a wide range of topologies to be expressed. The scripts also generate topology diagrams using `graphviz`. All topology diagrams in §3.7 were automatically generated.

Defaults.

- Router defaults:
 - Multicast is enabled by default and can be turned off globally or per router.
 - Two PIM (Protocol Independent Multicast) modes are supported: *PIM Sparse Mode* (PIM-SM), used for *Any Source Multicast* (ASM), and *PIM Source-Specific Multicast* (PIM-SSM). PIM-SSM is used by default.
 - When PIM-SM is used, the *Rendezvous Point* (RP) ID and ASM prefix are configurable.
 - The router name prefix is also configurable, defaulting to `r`.
- Links defaults:
 - The default bandwidth of all links can be changed. The syntax follows `tc`'s `RATE` format [66] and defaults to 1 Gbits/s.
 - The loss rate is configurable. The `loss` key sets the loss probability for each packet, and the `burst` key is a correlation percentage (i.e., each packet's loss probability depends on `burst` % of the previous one [66]).
 - The default link delay is configurable using `tc`'s `TIME` syntax and defaults to 1 millisecond.
 - `buffer` is the maximum number of packets the *queueing discipline* (`qdisc`) can hold at any given time [66], defaulting to 100 000.
 - Multicast can be configured per interface. Disabling it on a link means PIM will not be activated on the two interfaces on each side of the link. Defaults to true.

Router section.

The number of routers can be expressed as a list (e.g., `router: ["r1", "r2"]`) or as a number, with each router's name derived from the router's name prefix and its ID.

PIM can be disabled per router using the `overrides` key.

Router links are defined as regular links, except that they are assigned a different subnet range. Multicast (PIM) can also be disabled per link, in which case PIM will not be enabled on either interface of that link.

A link can also be marked as `GRE` (Generic Routing Encapsulation, RFC 2784 [17]). In that case, a virtual ethernet (veth) pair acting as an underlay will be created, along with a GRE tunnel interface on each node over which multicast traffic will flow.

All other parameters are configurable per link (delay, buffer size, bandwidth, loss percentage, burst loss percentage).

Servers and clients section.

Servers can only be specified as a list of names. Clients can be specified either as a list of names or as a number, with each client's name formed from the prefix "client" followed by its ID.

Links section.

The links between clients or servers and the routers can be configured in the same manner as router links, with per-link control over multicast, loss, delay, and buffer size.

A.3.0.1 Example topology file

yaml

```
1 topology:
2   defaults:
3     # router defaults
4     routers:
5       multicast: true
6       use_asm: false
7       rp_id: 1
8       asm_prefix: "224.0.0.0/4"
9       name_prefix: "r"
10
11     # link defaults
12     links:
13       bandwidth: "1Gbit"
14       loss: "0%"
15       burst: "0%"
16       delay: "1ms"
17       buffer: 100000
18       multicast: true
19
20   # List or number of routers
21   routers:
22     num: 3
23     overrides:
24       r3:
25         multicast: false
26     links:
27       - endpoints: ["r1", "r2"]
28         multicast: false
29         gre: true
30         delay: 10ms
31       - endpoints: ["r1", "r3"]
32         delay: 10ms
33
34   # List of servers
35   servers: ["server1"]
36
37   # List or number of clients
38   clients:
39     num: 3
40
41   links:
42     - endpoints: ["r1", "server1"]
43       delay: 1ms
44     - endpoints: ["r1", "client1"]
45     - endpoints: ["r2", "client2"]
46       loss: 1%
47       burst: 25%
48     - endpoints: ["r3", "client3"]
```

Listing 1: Example topology file structure

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl