muMuQUIC: Micro-Multicast for QUIC

Design and Hackathon Results

Louis Navarre
UCLouvain
louis.navarre@uclouvain.be
ORCID 0000-0001-8533-5667

Olivier Bonaventure
UCLouvain & WEL Research Institute
olivier.bonaventure@uclouvain.be
ORCID 0000-0002-6717-0296

14 September 2026

Abstract

Delivering the same content to many receivers over QUIC currently requires either one unicast connection per receiver or a substantial protocol extension. Flexicast QUIC solves the problem by building on Multipath QUIC, but in doing so it inherits the full multipath machinery — path probing, per-path nonce computation, and the rest — much of which flexicast delivery never exercises. This report asks how much of that machinery can be discarded, and how much must be reinvented in its place.

The answer is muMuQUIC ("micro-multicast QUIC"), a roughly 320-line specification. A server sends ordinary 1-RTT short-header packets to an IP source-specific multicast group. The only departures from a unicast packet are that the Destination Connection ID carries a Flow ID and that packet protection uses a flow-specific key, derived from a shared secret by the *unmodified* RFC 9001 procedure. Clients learn the group address, port, Flow ID, cipher suite and secret over their existing unicast connection through a single new frame, then feed multicast packets into their normal receive pipeline. Only DATAGRAM frames are carried. muMuQUIC deliberately provides no reliability, no congestion control and no source authentication.

The design was validated at the IETF 126 Hackathon in Vienna, where participants implemented it in six independent QUIC stacks in under 1.5 days and used it to deliver a live RTP video stream from a webcam to a laptop and two smartphones.

Disclaimer. This document was generated with the assistance of an AI language model, which reformatted the muMuQUIC design document into report form and drafted the introduction, results and status sections from an author-written blog post. The original source text is the `README.md` of https://github.com/louisna/minimal-multicast-quic-ietf-126, which remains authoritative in case of any discrepancy with this report.

1 Introduction

Flexicast QUIC [7, 8] extends QUIC and its multiple-path extension to support flexible multicast delivery, allowing a server to combine unicast and multicast within a single QUIC connection. The design is clean, but it requires the whole Multipath QUIC machinery to work: an implementation that does not already support multipath cannot support Flexicast QUIC without first acquiring it.

At the same time, several features of the multipath extension are not needed for flexicast delivery at all. Path probing and per-path nonce computation are two examples: a multicast flow has no return path to probe, and its own key and packet-number space already guarantee nonce uniqueness. This raises a concrete research question, and one with practical consequences for adoption: can Multipath QUIC be removed from the design entirely? Some parts would inevitably have to be reinvented — but how much, exactly?

We took that question to the IETF 126 Hackathon in Vienna, Austria, in July 2026, with a simple objective: determine the minimal set of features that must be added to QUIC to enable working multicast delivery in the protocol.

The answer is muMuQUIC, standing for micro-multicast QUIC. In roughly 320 lines of specification text, it describes a minimal extension that delivers unreliable data (DATAGRAM frames [9]) efficiently over multicast in QUIC.

Concretely, muMuQUIC is a minimal, **unreliable** extension that lets a QUIC server send DATAGRAM data to an IP multicast group, which multicast-capable clients receive as part of their existing unicast QUIC connection. The design goal is **the smallest possible delta** to an existing QUIC stack, and **interoperability** across independent implementations. Everything that is not strictly required to "put QUIC packets on a multicast address and have a client decode them" is deliberately left out. This work builds upon Flexicast QUIC, distilling it into a minimal proof of concept; for the full design, see the Flexicast QUIC IETF draft [7], the SIGCOMM CCR paper [8], and the open-source implementation [4].

The remainder of this report presents the design (Sections 2–12) and then the hackathon results (Section 13) and the current standardisation status (Section 14).

2 Scope and non-goals

As a first step, this document focuses on a **single multicast flow** per connection. Supporting multiple concurrent flows (e.g. the same content at different bit-rates) is a straightforward extension but is intentionally left out to keep the initial PoC minimal.

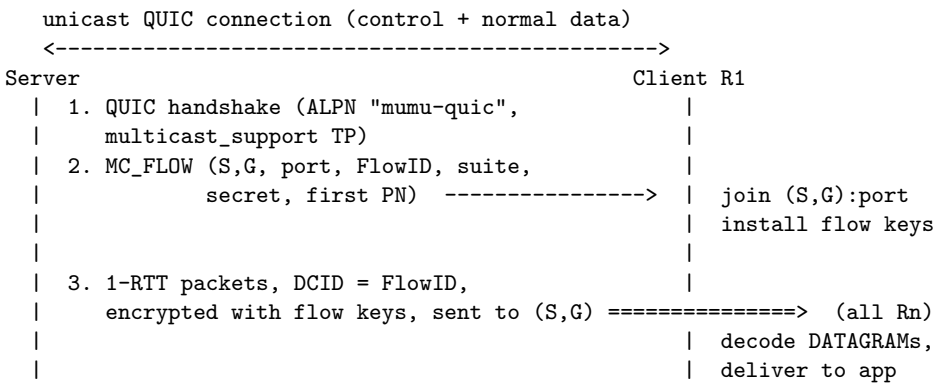
In scope:

- A server sends ordinary QUIC 1-RTT packets to an IP multicast (S,G) group.
- Those packets carry DATAGRAM frames (RFC 9221 [9]) only.
- A client, over its normal unicast QUIC connection, is told the group address, UDP port, a Flow ID, a cipher suite, and a secret; it joins the group and decodes the packets as if they had arrived over unicast.

Explicitly **out of scope** for this PoC:

- **No reliability**: no acknowledgements, no retransmission, no loss recovery for multicast data.
- **No Multipath QUIC.** The cleanest approach would be to rely on Multipath QUIC, as it solves numerous problems (e.g., unicast fallback and retransmission), but requires support of the extension in the underlying implementation. As such, this PoC does not use Multipath QUIC to increase the potential coverage of implementations. The multicast traffic is *not* an MP-QUIC path. It is an alternate packet input demultiplexed into the existing connection by DCID.
- **No flow control or congestion control** for the multicast traffic. The server sends at an application-chosen rate.
- **No integrity / source authentication** beyond AEAD with a shared key (see Section 12).
- **No key update / key rotation.** One secret per flow for its lifetime.
- **No join/leave/state signalling from the client.** The server does not track per-receiver flow membership.
- **No STREAM frames** on multicast (would pull in offset tracking and flow control).

3 Overview



Key idea: the packets on the multicast wire are **normal QUIC 1-RTT short-header packets**. The only differences from a unicast packet are (a) the Destination Connection ID is a **Flow ID**, and (b) they are protected with a **flow-specific key**. This lets the client reuse its existing receive → remove-header-protection → decrypt → parse-frames → deliver pipeline essentially unchanged.

The same Flow ID and secret are sent to **every** multicast-capable client, so a single encrypted packet is decodable by all of them.

4 Transport parameter

A single transport parameter negotiates support:

- `multicast_support` (hackathon value `0xff4d40`): **zero-length**. Presence in a client's transport parameters indicates the client can receive multicast flows and process the `MC_FLOW` frame.

A server MUST NOT send `MC_FLOW` frames to a client that did not send `multicast_support`. A client that did not send `multicast_support` MUST treat receipt of an `MC_FLOW` frame as a connection error of type `PROTOCOL_VIOLATION`.

Note: For the PoC, implementations may decide to skip the aforementioned treatment of protocol violation.

Note: For the PoC, only the client → server direction matters, since the server is the only sender of multicast data.

5 The MC_FLOW frame

`MC_FLOW` is sent **server → client**, on the unicast connection, in a 1-RTT packet. It combines "here is a flow" and "here is its key" into one frame, which is possible because there is no reliability and no key rotation to sequence. This is a difference with existing drafts.

```
MC_FLOW Frame {
  Type (i) = 0xff4d43,
  Flow ID Length (8),
  Flow ID (8..160),
  IP Version (8),
  Source IP (32 or 128),
  Group IP (32 or 128),
  UDP Port (16),
  Cipher Suite (16),
  First Packet Number (i),
  Secret Length (i),
```

```
  Secret (..),
}
```

Fields:

- **Flow ID Length**: length in bytes of the Flow ID. MUST be between 1 and 20. A value of 0 or > 20 MUST be treated as `FRAME_ENCODING_ERROR`.
- **Flow ID**: the connection ID that appears as the DCID of packets on this flow. MUST NOT be zero-length. The client MUST NOT use this value as a connection ID for its unicast connection; if it is already in use, the client retires it first.

  Note: For this PoC, implementations may assume that the Flow ID always lies within the [1, 20] length, and that it is not already used by the client (low probability).
- **IP Version**: 4 or 6. Any other value is `FRAME_ENCODING_ERROR`. Selects the width of the two address fields (32 bits for IPv4, 128 bits for IPv6).
- **Source IP**: the multicast source address S (network byte order).
- **Group IP**: the multicast group address G (network byte order). MUST be a valid SSM destination address (RFC 4607 [2]).
- **UDP Port**: destination UDP port for the flow (network byte order).
- **Cipher Suite**: a TLS cipher suite code point (see Section 10). Determines the AEAD, the header-protection algorithm, and the hash used for key derivation.
- **First Packet Number**: the packet number the client should use as the initial "next expected" value for packet-number reconstruction, before it has decrypted any packet on the flow. This value is necessary to decrypt incoming packets if the client joins the flow later.
- **Secret / Secret Length**: the flow traffic secret from which packet protection keys are derived (Section 6). Its length is the hash length of the cipher suite.

`MC_FLOW` is ack-eliciting on the unicast connection and retransmitted using normal QUIC loss recovery until acknowledged.

Optional withdraw (not required for the PoC): a server MAY signal flow teardown by sending an `MC_FLOW` with `Secret Length = 0` for the same Flow ID; on receipt the client leaves the group and drops flow state. If omitted, the client simply leaves the group when the unicast connection closes.

6 Flow key derivation

The Secret plays exactly the role of a QUIC 1-RTT application traffic secret. Keys are derived with the **unmodified** RFC 9001 [11] procedure, using the hash and AEAD of the cipher suite:

```
flow_key = HKDF-Expand-Label(Secret, "quic key", "", key_len)
flow_iv  = HKDF-Expand-Label(Secret, "quic iv",  "", iv_len)
flow_hp  = HKDF-Expand-Label(Secret, "quic hp",  "", hp_len)
```

Because this is identical to normal 1-RTT key derivation, an implementation can call its **existing** "install 1-RTT keys from this secret" function, tagging the result as the flow's key context. The server generates the Secret randomly (hash-length bytes) once per flow.

7 Multicast packet format

Packets on the flow are standard 1-RTT short-header packets (RFC 9000 [5] Section 17.3.1):

- Header Form = 0 (short header), Fixed Bit = 1.
- **Spin Bit = 0**; clients ignore it.
- **Key Phase = 0**, fixed (there is no key update).
- **Packet Number Length**, indicating the packet number, similarly to QUIC packet format.

  Note: This choice is subject to discussion if the `First Packet Number` field of the `MC_FLOW` is sufficient to recover the true packet number, as done in the implementations of Flexicast QUIC.
- Destination Connection ID = the **Flow ID**.
- Packet payload contains only `DATAGRAM` frames, optionally with `PADDING` / `PING`. No other frame types are sent on the flow; a client MUST ignore any other frame type received on the flow.

Packet protection (AEAD nonce = left-padded packet number XOR `flow_iv`) and header protection use the flow key context and are otherwise unchanged from RFC 9001 [11]. Since the flow has its own key and its own packet-number space, nonce uniqueness holds without any Path ID or multipath construct.

Packet number reconstruction: the flow has its own packet-number space. The client reconstructs full packet numbers (RFC 9000 [5] Section 17.1) using the largest packet number it has successfully decrypted on the flow as the basis; before the first successful decrypt it uses `First Packet Number` from `MC_FLOW`. The server sends flow packet numbers continuously starting from `First Packet Number`.

8 Client processing

On receiving `MC_FLOW`:

1. Derive the flow key context (Section 6).
2. Open a UDP socket bound to the flow UDP port and **join the (S,G)** using the platform source-specific-multicast join (e.g. `MCAST_JOIN_SOURCE_GROUP` / `IP_ADD_SOURCE_MEMBERSHIP`).
3. Register the Flow ID so that incoming short-header packets whose DCID equals it are routed to this connection's flow key context and flow packet-number space.

On receiving a packet on the multicast socket: feed it into the normal QUIC receive routine. Because its DCID is the Flow ID, it is decrypted with the flow keys, its packet number is reconstructed in the flow space, and its `DATAGRAM` frames are delivered to the application exactly like unicast datagrams.

Two behaviours differ from a normal received packet:

- The client MUST NOT acknowledge flow packets (there is no return path and no reliability). The receive path for flow packets skips all ack-eliciting / ACK-generation bookkeeping.
- Flow packets MUST NOT reset the unicast connection's idle timer. Liveness is determined solely by the unicast connection.

When the unicast connection closes, the client leaves the (S,G) and discards flow state.

9 Server processing

1. When a multicast-capable client's connection is established, send an `MC_FLOW` frame on its unicast connection describing the flow (the **same** Flow ID and Secret for all clients sharing the flow).
2. Maintain one flow context: the derived keys, `flow_iv`, and a single monotonic packet-number counter starting at `First Packet Number`.
3. To transmit application data, build a 1-RTT short-header packet with DCID = Flow ID and PN = counter++, place the data in `DATAGRAM` frame(s), protect it with the flow keys, and `sendto` the (S,G):port. One packet reaches all receivers.

The server keeps no per-receiver state for the flow and processes no feedback for it.

10 Interop profile

To eliminate interop friction between quic-go, quiche, and aioquic, all endpoints in the hackathon MUST use these exact choices:

- ALPN = `"mumu-quic"`, offered by the client and selected by the server during the TLS handshake.
- Transport parameter `multicast_support` = `0xff4d40` (zero-length).
- Frame type `MC_FLOW` = `0xff4d43`.
- Cipher suite = `TLS_AES_128_GCM_SHA256` (`0x1301`): 16-byte key, 12-byte IV, 16-byte header-protection key, 32-byte secret. (The Cipher Suite field remains in the frame for generality, but only this value is exercised.)
- Packet number encoded as in RFC 9000 [5] Section 17.1 (1–4 bytes, length carried in the Packet Number Length bits), Key Phase 0, on all flow packets.
- Payload is `DATAGRAM` frames only.
- IPv4 first; IPv6 optional if time allows.

These codepoints are arbitrary experimental values; their only requirement is that every implementation uses the same ones.

11 What you reuse vs. what you add

Reused unchanged (no new code):

- TLS handshake and the whole unicast connection.
- Key derivation from a secret (fed the flow Secret).
- AEAD packet protection + header protection.
- Short-header packet parsing.
- `DATAGRAM` frame parsing and delivery to the application.

New code (the entire delta), per side:

Client:

1. Advertise / parse the `multicast_support` transport parameter.
2. Parse the `MC_FLOW` frame.
3. Open + source-join the multicast UDP socket.
4. Register Flow ID → flow key context + flow PN space; derive keys via the existing routine.
5. Route packets with DCID = Flow ID to the flow context and suppress ACK/idle-timer bookkeeping for them.

Server:

1. Emit / parse the transport parameter.
2. Generate the flow secret, derive keys (existing routine), open the multicast socket.

3. Emit `MC_FLOW` on each capable client's unicast connection.
4. Send path: build, protect, and `sendto` flow packets with the flow DCID, keys, and PN counter.

12 Security considerations

This PoC intentionally provides no source authentication and no integrity beyond AEAD with a shared key. Every receiver holds the same flow key, so any receiver (or anyone given the key) can forge packets to the group that other receivers will accept. There is no protection against a malicious group member injecting data.

This is acceptable only for a controlled/lab hackathon environment. A real deployment needs packet-level authentication independent of the shared key (e.g. the `MC_INTEGRITY` mechanism of draft-jholland-quic-multicast [3] and the analysis in draft-krose-multicast-security [10]). That machinery is deliberately excluded here to keep the change minimal.

13 Results

In less than 1.5 days of work at the IETF 126 Hackathon, participants implemented muMuQUIC in **six different QUIC stacks**, including Cloudflare quiche, Quinn, aioquic, and quic-go. Two of these serve as the reference implementations for this report: a quiche fork [6] and a Quinn implementation [1].

As a proof-of-concept application, we used the FFmpeg toolkit to generate an RTP stream from a webcam and delivered it over muMuQUIC DATAGRAM frames. Participants could then receive the RTP stream either through GStreamer or through custom RTP parsers, once the muMuQUIC DATAGRAM frames had been processed. The stream was displayed simultaneously on a laptop and on two smartphones running muMuQUIC.

The transport medium deserves a precise statement. The video frames traversed the IETF hotel Wi-Fi network, but this was *not* native multicast over the air: the access point performed multicast-to-unicast conversion. The efficiency gain from multicast is therefore realised upstream of the wireless hop, not on it. With that caveat, the demonstration worked: a single server-side packet stream fed all receivers.

14 Status and next steps

This result was presented at the QUIC Working Group meeting at IETF 126, jointly with Max Franke (TU Berlin), who also works on multicast extensions for QUIC. The community feedback was positive, and led to a non-WG-forming Birds of a Feather (BoF) session scheduled for IETF 127 in San Francisco.

Acknowledgements

We thank Vany Ingenzi, Anthony Doeraene, François Michel, Maxime Piraux, and Max Franke for their contributions to the muMuQUIC implementations and to the hackathon effort.

References

[1] Anthony Doeraene. quinn-minimal-mcquic: A muMuQUIC implementation in Quinn. Software repository, 2026. URL https://github.com/Aperence/quinn-minimal-mcquic.

[2] Hugh Holbrook and Brad Cain. Source-specific multicast for IP. RFC 4607, IETF, August 2006. DOI 10.17487/RFC4607.

[3] Jake Holland, Lucas Pardue, Max Franke, and Kyle Rose. Multicast extension for QUIC. Internet-Draft draft-jholland-quic-multicast-09, IETF, July 2026. URL https://datatracker.ietf.org/doc/draft-jholland-quic-multicast/. Work in Progress.

[4] IP Networking Lab. Flexicast QUIC: A Quiche fork implementing flexicast QUIC. Software repository, 2025. URL https://github.com/IPNetworkingLab/flexicast-quic.

[5] Jana Iyengar and Martin Thomson. QUIC: A UDP-based multiplexed and secure transport. RFC 9000, IETF, May 2021. DOI 10.17487/RFC9000.

[6] Louis Navarre. quiche-minimal-multicast: A muMuQUIC implementation in Cloudflare Quiche. Software repository, 2026. URL https://github.com/louisna/quiche-minimal-multicast.

[7] Louis Navarre and Olivier Bonaventure. Flexicast QUIC: Combining unicast and multicast in a single QUIC connection. Internet-Draft draft-navarre-quic-flexicast-03, IETF, September 2026. URL https://datatracker.ietf.org/doc/draft-navarre-quic-flexicast/. Work in Progress.

[8] Louis Navarre, Quentin De Coninck, Tom Barbette, and Olivier Bonaventure. Taking the best of multicast and unicast with flexicast QUIC. *ACM SIGCOMM Computer Communication Review*, 55(2):2–12, April 2025. ISSN 0146-4833. doi: 10.1145/3750832.3750834.

[9] Tommy Pauly, Eric Kinnear, and David Schinazi. An unreliable datagram extension to QUIC. RFC 9221, IETF, March 2022. DOI 10.17487/RFC9221.

[10] Kyle Rose, Max Franke, and Jake Holland. Security and privacy considerations for multicast transports. Internet-Draft draft-krose-multicast-security-07, IETF, May 2025. URL https://datatracker.ietf.org/doc/draft-krose-multicast-security/. Work in Progress.

[11] Martin Thomson and Sean Turner. Using TLS to secure QUIC. RFC 9001, IETF, May 2021. DOI 10.17487/RFC9001.