Media Over QUIC
Internet-Draft
Intended status: Standards Track
Expires: 1 September 2026

S. Nandakumar
Cisco
C. Jennings
Cisco Systems

28 February 2026

ATOM: AT Protocol Over MoQ Transport
draft-nandakumar-atproto-atom-00

Abstract

This document specifies how the Authenticated Transfer (AT) Protocol can leverage Media over QUIC Transport (MOQT) for efficient data synchronization across decentralized social networks. The AT Protocol's firehose event stream and repository synchronization mechanisms map naturally to MOQT's publish/subscribe model, enabling scalable relay infrastructure, priority-based delivery, and improved resilience for large-scale social data distribution.

This specification addresses the challenges of the current WebSocket-based transport and demonstrates how MOQT's relay architecture, group-based caching, and multiplexed streams provide significant benefits for AT Protocol deployments at scale.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://snandaku.github.io/draft-nandakumar-atproto-atom/draft-nandakumar-atproto-atom.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-nandakumar-atproto-atom/.

Discussion of this document takes place on the Media Over QUIC Working Group mailing list (mailto:moq@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/moq/. Subscribe at https://www.ietf.org/mailman/listinfo/moq/.

Source for this draft and an issue tracker can be found at https://github.com/snandaku/draft-nandakumar-atproto-atom.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Authenticated Transfer (AT) Protocol [AT-ARCH] provides a framework for decentralized social web applications using self-certifying data repositories. The protocol enables users to maintain control over their data while participating in a federated network of Personal Data Servers (PDS), relays, and application views (AppViews).

Currently, AT Protocol relies on two primary transport mechanisms for data synchronization:

* *HTTP*: Used for batch repository exports via CAR (Content Addressable Archive) files

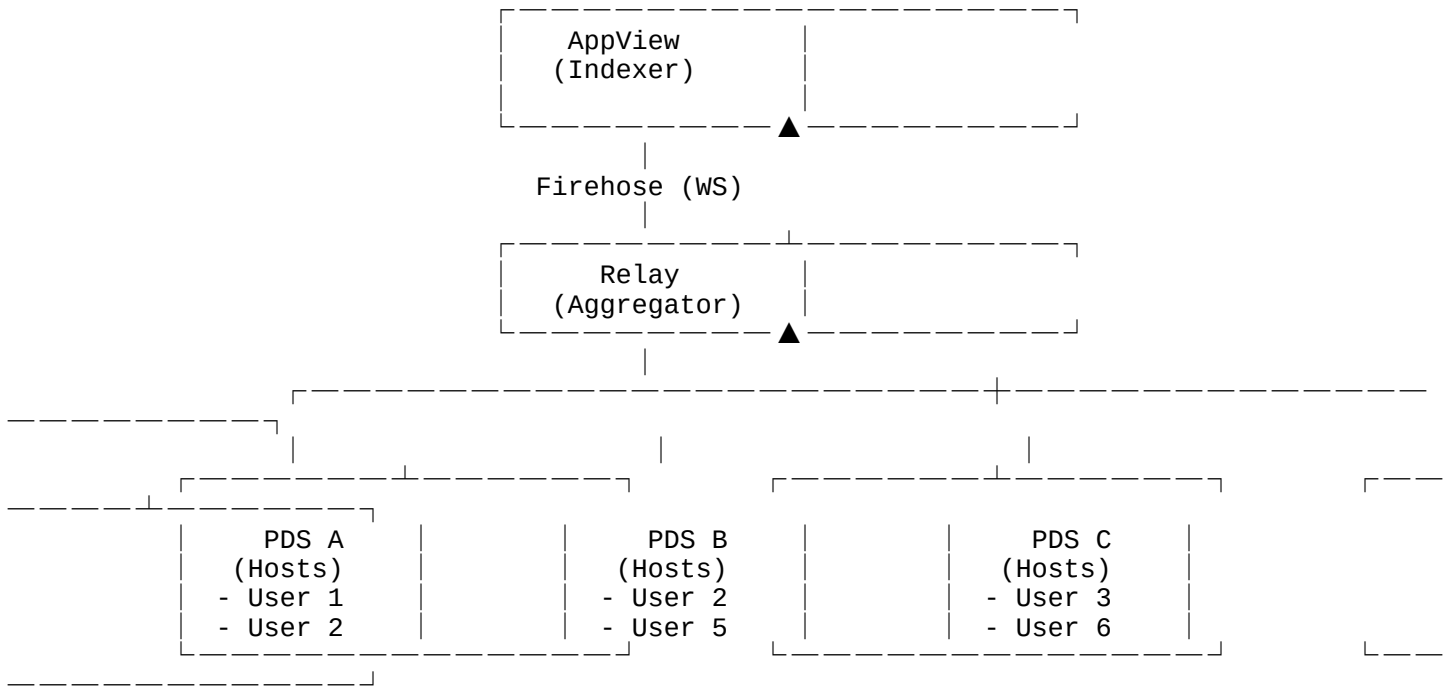
* *WebSocket*: Used for real-time event streams (firehose) with CBOR-encoded messages

While functional, these transport mechanisms face significant challenges at scale, particularly for the firehose event stream that must distribute updates across a network with millions of users generating billions of records.

This document defines how Media over QUIC Transport (MOQT) [MoQ-TRANSPORT] can serve as an enhanced transport for AT Protocol. MOQT enables low-latency delivery while relay caching supports multiple latency spectrums from real-time to eventual consistency. The protocol's native publish/subscribe model, relay infrastructure, and priority-based delivery address current scalability and resilience challenges.

1.1. Current AT Protocol Architecture

AT Protocol Network Architecture:



The current architecture comprises:

Personal Data Server (PDS): Hosts user accounts and their data repositories. Each PDS provides a firehose endpoint for real-time updates and HTTP endpoints for repository exports.

Relay: Aggregates firehose streams from multiple PDS instances into a unified event stream. Full-network relays attempt to include all PDS instances in the network.

AppView: Application-specific indexing services that consume the firehose to build aggregated views (e.g., feeds, search, analytics).

1.2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Challenges with Current Transport

The current WebSocket-based transport for AT Protocol faces several challenges that impact scalability, reliability, and operational efficiency.

2.1. Scalability Limitations

2.1.1. Connection Overhead

Each subscriber to a PDS or relay firehose requires a dedicated WebSocket connection. At network scale, this creates:

* Connection state overhead on servers hosting popular content

* TCP head-of-line blocking affecting all events on a connection

* Limited ability to prioritize critical events during congestion

2.1.2. Message Size Constraints

The current protocol specifies a hard maximum of 5 MBytes per WebSocket frame. While this accommodates most operations, it creates challenges for:

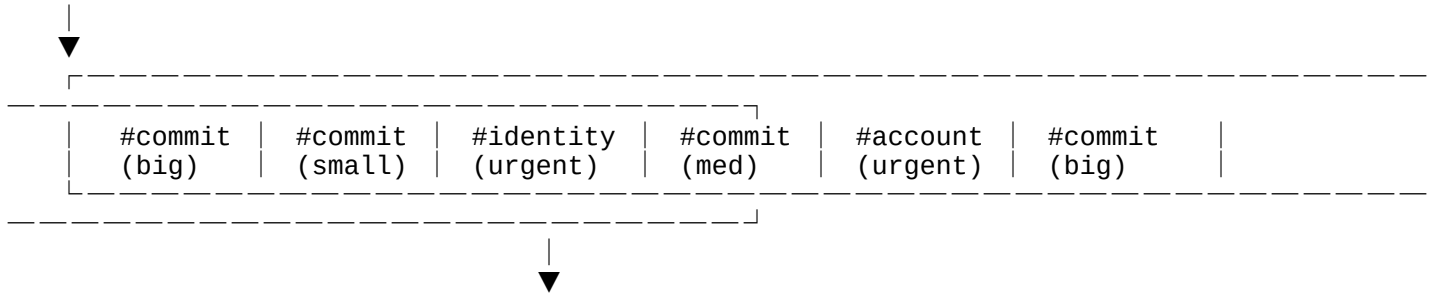
* Large repository commits (limited to 2MB blocks, 200 operations)

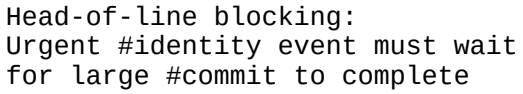
* Efficient batching of small updates

* Variable-size content distribution

2.1.3. Single-Stream Bottleneck

Current Firehose: Single Stream for All Event Types





All event types flow through a single WebSocket stream without differentiation, meaning:

* High-priority identity changes wait behind large commit events

* Account status updates cannot preempt in-progress transfers

* No mechanism for subscribers to filter by event type at transport layer

2.2. Reliability Challenges

2.2.1. Cursor-Based Replay Limitations

The current firehose relies on monotonic cursor values for replay after disconnection. This approach requires consumers to track cursor state externally, since the protocol provides no mechanism for server-side cursor persistence. Gap detection depends entirely on commit chain validation, which is computationally expensive and requires maintaining local state. When gaps are detected, recovery requires fetching the complete repository, creating a thundering herd risk when many consumers reconnect simultaneously after an outage.

2.2.2. No Native Late-Join Support

When a new consumer connects to a firehose, it must either start from the current position and miss all prior events, or replay from a cursor to catch up. There is no concept of "recent state" that would allow quick synchronization to a meaningful starting point. Consumers cannot receive cached recent events without initiating a full replay sequence, which increases latency for new subscribers and places additional load on the server.

2.2.3. Connection Fragility

WebSocket connections over TCP are susceptible to network path changes that cause connection drops. The protocol provides no mechanism for connection migration when a client’s IP address changes, such as when moving between networks on a mobile device. Any network transition requires establishing a completely new connection and resuming from the last known cursor position.

2.3. Operational Challenges

2.3.1. Relay Infrastructure Complexity

Current relays face significant operational burden in maintaining the firehose infrastructure. Each relay must establish and maintain persistent WebSocket connections to all upstream PDS instances it aggregates, scaling linearly with the number of sources. When connections fail, relays must handle reconnection logic and cursor tracking independently for each upstream source. Custom caching and replay logic must be implemented at the application layer, as the transport provides no native support for these operations. Additionally, relays must build proprietary distribution mechanisms to serve their own downstream subscribers, duplicating effort across the ecosystem.

2.3.2. Limited Quality of Service

The current transport provides no native mechanism for quality of service differentiation. All event types receive equal treatment

regardless of their operational importance, so critical account status changes compete with routine content updates for bandwidth. Rate limiting based on consumer capacity must be implemented entirely at the application layer, with no transport-level flow control. Graceful degradation during overload conditions is difficult to achieve, often resulting in connection failures rather than controlled service reduction.

3. MOQT Transport Benefits

MOQT addresses the challenges identified above through its native architecture designed for large-scale media distribution.

3.1. Native Publish/Subscribe Model

MOQT's publish/subscribe model aligns naturally with AT Protocol's firehose semantics. Subscribers select specific event tracks of interest while publishers announce available tracks via PUBLISH_NAMESPACE. Since relays handle distribution, no dedicated connection is required per subscriber, and track-level subscription granularity reduces unnecessary traffic.

PUBLISH_NAMESPACE enables dynamic discovery of available content. When a PDS comes online, it announces its namespaces to connected relays, advertising firehose tracks, repository sync tracks, and blob stores. Relays learn about new PDS instances automatically without static configuration, enabling the network to scale organically as new servers join.

SUBSCRIBE_NAMESPACE allows subscribers to discover available tracks using prefix-based queries. A full-network relay can subscribe to at/firehose/ to discover all PDS instances publishing firehose events, then selectively subscribe to each. An AppView can query a relay's namespace to enumerate available event types before subscribing to specific tracks of interest. This discovery mechanism eliminates the need for out-of-band coordination when adding new publishers or subscribers to the network.

A PDS advertises its available namespaces via PUBLISH_NAMESPACE:

Namespace	Description
at/firehose/pds.example.com	Firehose events from this PDS
at/repo/pds.example.com	Repository sync tracks
at/blobs/pds.example.com	Blob storage

Table 1

A relay discovering PDS instances issues SUBSCRIBE_NAMESPACE with at/firehose/ and receives a NAMESPACE response listing all matching hosts (potentially thousands of PDS instances).

3.2. Priority-Based Delivery

MOQT's 0-255 priority scale enables differentiated event delivery, ensuring that critical account takedowns (priority 0-15) and urgent identity changes such as key rotations (priority 16-31) are delivered ahead of routine commit events (priority 96-191). This prevents large content transfers from blocking time-sensitive operations.

AT Event Type	MOQT Priority	Rationale

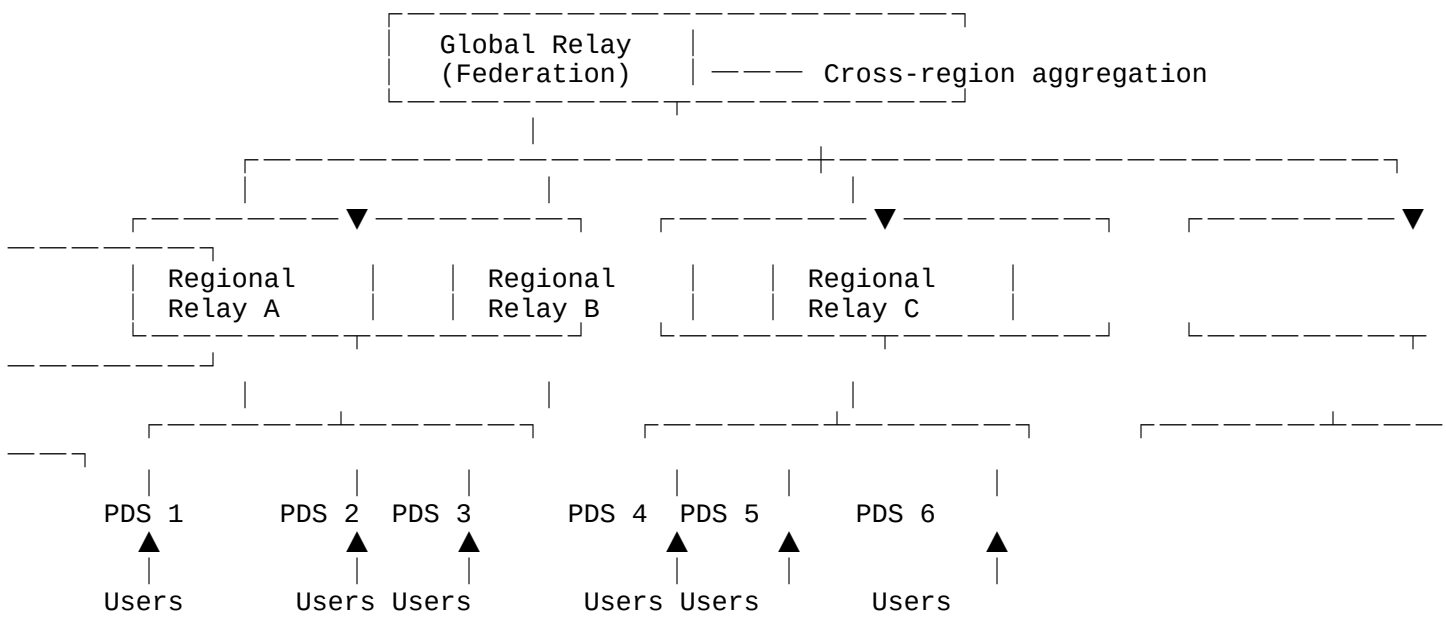
#account (takedown)	0-15	Critical - affects content availability
#identity	16-31	Urgent - key rotation, handle changes
#account (status)	32-63	Important - account state changes
#sync	64-95	Moderate - state reset events
#commit (small)	96-127	Normal - typical record operations
#commit (large)	128-191	Background - bulk content updates
Repository export	192-255	Bulk - full sync operations

Table 2

3.3. Relay Infrastructure

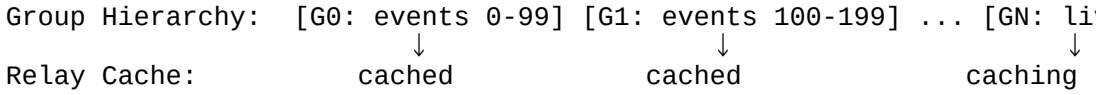
MOQT relays provide purpose-built infrastructure for AT Protocol distribution. Hierarchical relay topologies reduce origin load by distributing cached content from regional nodes, while geographic placement optimizes latency for subscribers. Because the relay protocol is standardized, operators need not implement custom distribution logic.

MOQT Relay Network for AT Protocol:



3.4. Group-Based Organization and Caching

MOQT groups enable efficient caching and late-join support. Late-joining subscribers receive the most recent group immediately rather than replaying from the beginning, and relay caches at each tier reduce upstream load. Group boundaries serve as natural replay points, with cursor semantics mapping directly to Group ID and Object ID.



3.4.1. Late-Join via SUBSCRIBE

MOQT's SUBSCRIBE message includes filter parameters that control the starting position for delivery. As defined in [MoQ-TRANSPORT], these filters include LatestGroup (start from the newest available group), LatestObject (start from the most recent object), AbsoluteStart (begin at a specific group/object position), and AbsoluteRange (request a bounded range). These filters allow subscribers to join at any point and then continue receiving live and future content.

SUBSCRIBE delivers content from the specified starting position forward, keeping subscribers at the live edge as new content arrives. For historical content retrieval, subscribers use FETCH operations instead (see Section 3.4.2).

This capability supports several AT Protocol scenarios. A new subscriber joining a track uses LatestGroup to begin with current activity.

3.4.2. FETCH for On-Demand Retrieval

MOQT's FETCH operation complements SUBSCRIBE by enabling selective retrieval of specific objects. While SUBSCRIBE keeps clients at the live edge, FETCH retrieves historical content regardless of how far back it exists. FETCH specifies exact group and object ranges, retrieving precisely the needed data without establishing a continuous subscription.

When a subscriber detects a gap in received events, it can FETCH the missing objects directly rather than replaying an entire group. When verifying a specific record, a client can FETCH just the MST path blocks needed for cryptographic validation. For deep historical queries, an analytics service can FETCH specific commit ranges from months prior.

Relay caching amplifies the benefit of FETCH operations. Popular content such as viral images, high-profile account commits, and frequently accessed MST nodes remain in relay cache. Subsequent FETCH requests for this content are served locally, reducing latency and eliminating load on origin servers. For a viral post with an embedded image, thousands of FETCH requests for the image blob resolve to a single origin fetch followed by cache hits at the relay tier.

3.4.3. Caching Benefits Summary

The combination of SUBSCRIBE late-join and FETCH with relay caching provides several benefits for AT Protocol deployments:

* Fan-out efficiency: Origin publishes once, relay serves many subscribers
* Late-join speed: New subscribers receive cached history immediately
* Gap recovery: Missing events fetched from relay cache, not origin
* Popular content: High-demand blobs and commits served from cache
* Reduced origin load: Relay tier absorbs read amplification

3.5. QUIC Transport Advantages

Leveraging QUIC [RFC9000] provides connection migration so that firehose subscriptions survive network changes such as IP address transitions or WiFi-to-cellular handoffs. Multiplexed streams eliminate head-of-line blocking between logical channels, while mandatory TLS 1.3 [RFC8446] ensures built-in encryption. Per-stream and connection-level flow control prevents subscriber overload, and 0-RTT resumption enables fast reconnection after brief disconnections.

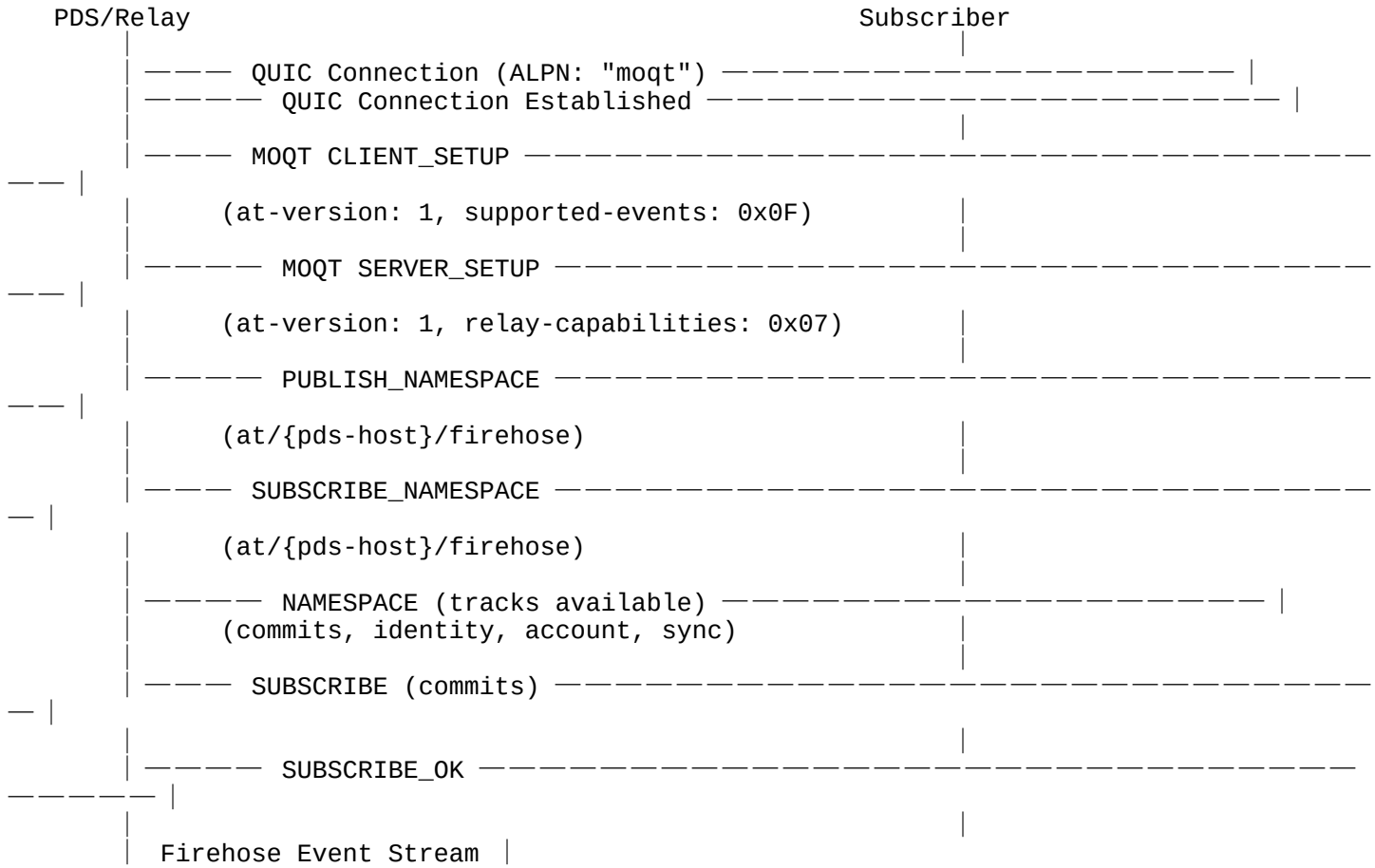
4. MOQT Mapping

This section defines how AT Protocol data maps to MOQT primitives.

4.1. Connection Establishment

AT Protocol endpoints using MOQT establish connections following the standard MOQT setup procedure:

AT-over-MOQT Connection Establishment:



Setup parameters for AT Protocol:

Parameter	ID	Type	Description
at-version	0x41540001	varint	AT Protocol version
at-supported-events	0x41540002	varint	Bitmask of supported event types
at-relay-caps	0x41540003	varint	Relay capability flags

Table 3

Event type bitmask values: 0x01 (#commit), 0x02 (#identity), 0x04 (#account), 0x08 (#sync).

4.2. Track Types

ATOM defines three track types for different use cases:

4.2.1. Firehose Tracks

Firehose tracks carry real-time event streams from PDS hosts.

Namespace structure: at/firehose/{host}/{event-type} -- {did|all}

Track	Namespace	Content	Priority
Commits	at/firehose/{host}/commits	Repository updates	96-191
Identity	at/firehose/{host}/identity	DID/handle changes	16-31
Account	at/firehose/{host}/account	Hosting status	0-63
Sync	at/firehose/{host}/sync	State assertions	64-95

Table 4

Groups organize events by sequence number (e.g., 1000 events per group). Track name is either a specific DID or "all" for aggregated streams.

4.2.2. Repository Sync Tracks

Repository sync tracks provide full block-level repository data.

Namespace: at/repo/{host}/{did}/sync

The MOQT hierarchy maps to AT Protocol repository concepts:

MOQT Level	AT Protocol Concept	Purpose
Track	Repository (DID)	Subscription unit
Group	Commit/Revision	Temporal progression
Subgroup	Collection + MST path	Structural organization
Object	Record/Block	Individual data items

Table 5

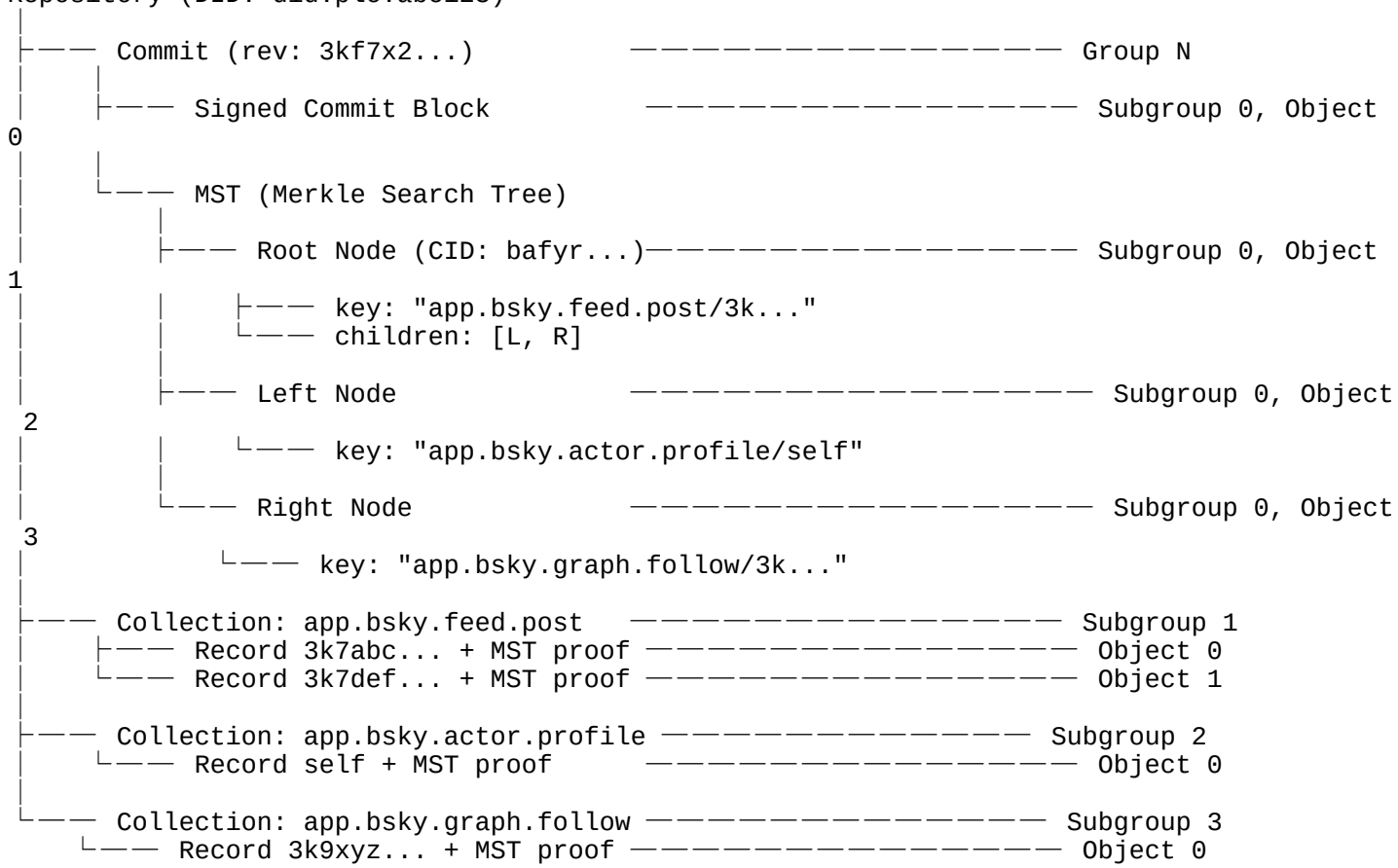
Each group represents a commit. Subgroup 0 contains the signed commit and MST structure; subsequent subgroups contain collections with MST proof paths for independent verification.

Subgroup	Contents
0	Commit block + full MST (verification anchor)
1+	Collection records + MST proof path

Table 6

Objects within a group are ordered: signed commit (ObjectID 0), MST root, MST intermediate nodes, then records. The at-block-cid extension header carries the CID for integrity verification.

MST Structure and MOQT Mapping:



The MST proof path included with each record contains the minimal set of tree nodes required to verify that record's inclusion in the commit. This enables independent verification of individual records without downloading the entire repository.

4.2.3. Blob Store Track

Blob store tracks enable cross-repository deduplication of large media.

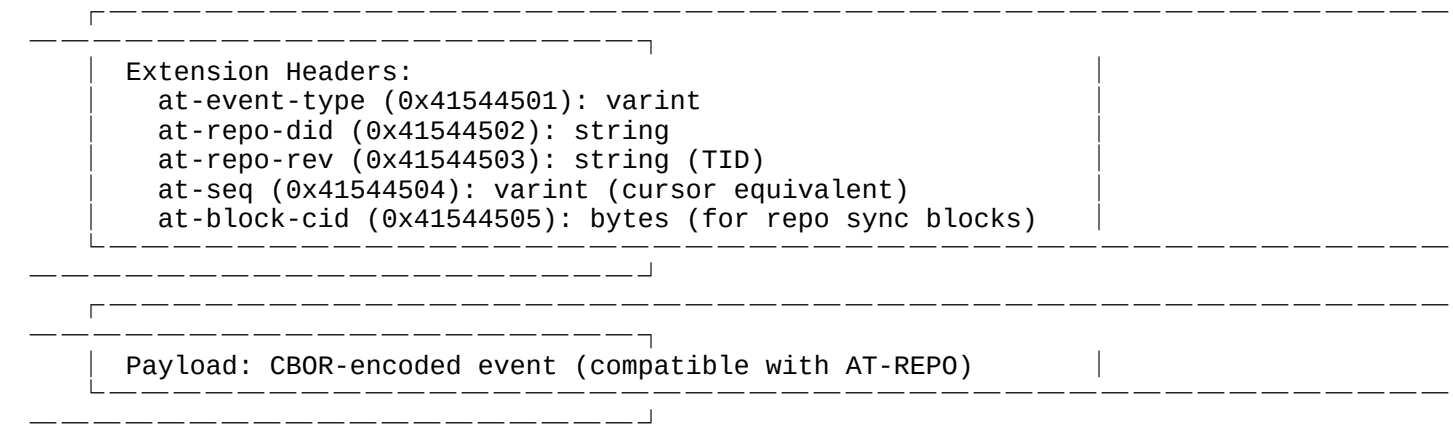
Namespace: at/blobs/{host}

The Group ID is derived from the CID (first 8 bytes of multihash as uint64), Object ID is always 0. This deterministic mapping ensures identical blobs from different repositories resolve to the same MOQT location.

4.3. Object Structure

AT Protocol events are encapsulated in MOQT objects:

MOQT Object Structure:



4.4. Subscription and Filtering

4.4.1. Firehose Subscription

Subscribers specify start position and filtering:

* StartGroup=Latest: Begin with live events (default)

* StartGroup=N: Resume from specific cursor position

* Track name all or specific DID for filtering

4.4.2. Repository Sync

Subgroup filtering enables selective synchronization:

Selective Sync Examples:

```
Full repository:     SubgroupFilter: [0, 1, 2, 3, ...]
Posts only:          SubgroupFilter: [0, 1]  (~15% bandwidth)
Verification only:   SubgroupFilter: [0]     (~2% bandwidth)
```

Temporal navigation with groups:

* Subscribe StartGroup=LATEST: Current state + live updates

* Fetch StartGroup=N, EndGroup=M: Historical range retrieval

4.4.3. Late-Join and Catch-Up

New subscribers request StartGroup=LATEST to receive current state immediately. After disconnection, subscribers use Fetch with their last known group to retrieve missed commits before resuming live subscription.

4.5. Priority and Parallelism

MOQT priority enables efficient delivery ordering:

Block Type	Priority	Rationale
Account takedown	0-15	Safety-critical
Identity changes	16-31	Affects routing

Signed Commit	0-15	Required for verification
MST Root	16-31	Unlocks tree traversal
MST Nodes	32-63	Enables record verification
Small Records	64-127	Core content
Large Records	128-191	Less urgent
Blobs	192-255	Media, lazy-loadable

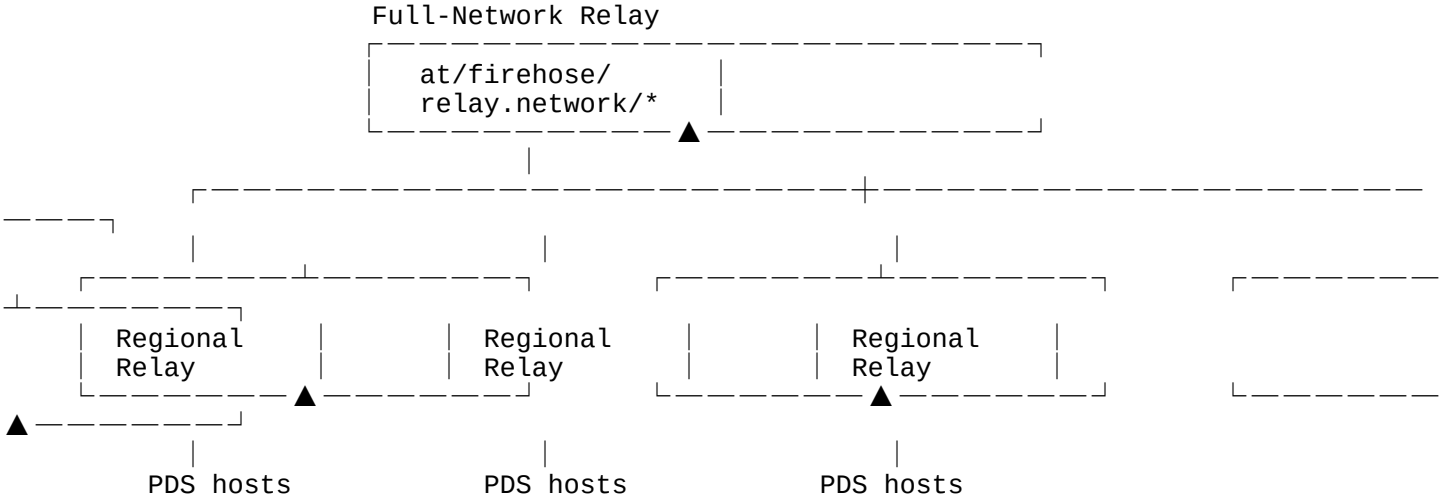
Table 7

Content-addressable blocks enable parallel fetching across QUIC streams, with total transfer time bounded by the slowest stream rather than the sum of all transfers.

4.6. Relay Aggregation

MOQT relays aggregate firehose streams from multiple upstream sources:

Relay Aggregation:



Relays perform subscription aggregation, namespace republishing, event caching for late-join, and deduplication.

5. Reliability and Recovery

ATOM leverages MOQT's group-based delivery model and QUIC's transport features to provide robust reliability mechanisms for AT Protocol data synchronization. The key capabilities include cursor mapping that translates AT Protocol sequence numbers to MOQT group/object positions, gap detection that identifies missing events through sequential object tracking, and multiple recovery strategies ranging from targeted object fetches to full repository synchronization. QUIC's connection migration further ensures that subscribers maintain continuous delivery even when network conditions change.

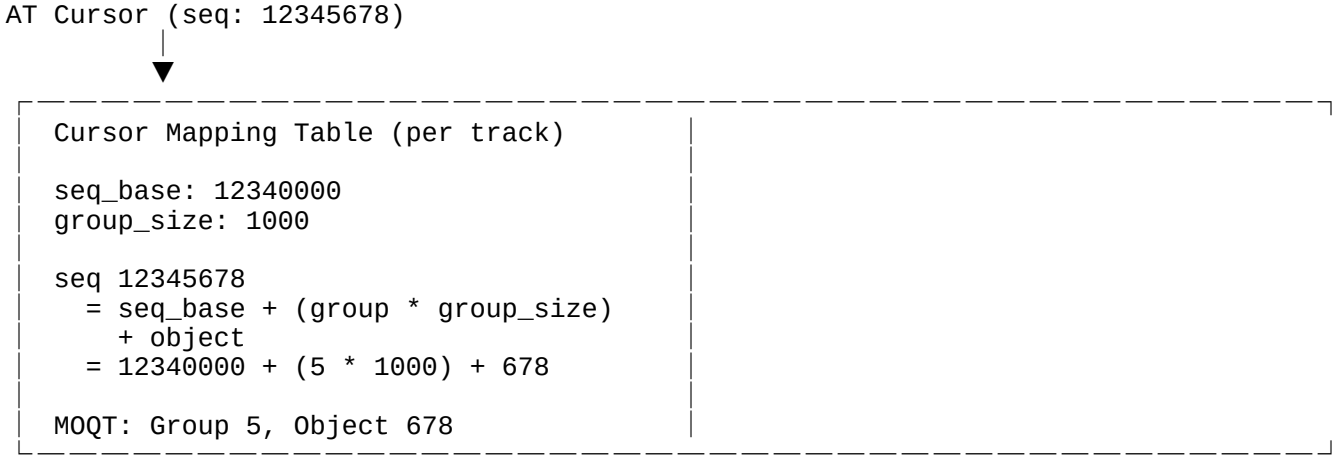
5.1. Cursor Mapping

AT Protocol cursors map to MOQT group and object identifiers, enabling seamless translation between the existing cursor-based replay mechanism and MOQT's hierarchical addressing. Each firehose track maintains a mapping table that records the sequence number base for each group, allowing bidirectional conversion between AT Protocol

cursors and MOQT positions.

When a subscriber needs to resume from a specific cursor position, it converts the cursor value to the corresponding group and object identifiers. The conversion uses integer division to determine the group number and the modulo operation to determine the object offset within that group. This deterministic mapping ensures that any cursor value can be precisely located within the MOQT track structure.

Cursor Translation:



Publishers MUST include the at-seq extension header in each object to enable accurate cursor reconstruction.

5.2. Gap Detection and Recovery

MOQT's group-based delivery model simplifies gap detection compared to the current cursor-only approach. This section describes how gaps are detected and the recovery strategies available to subscribers.

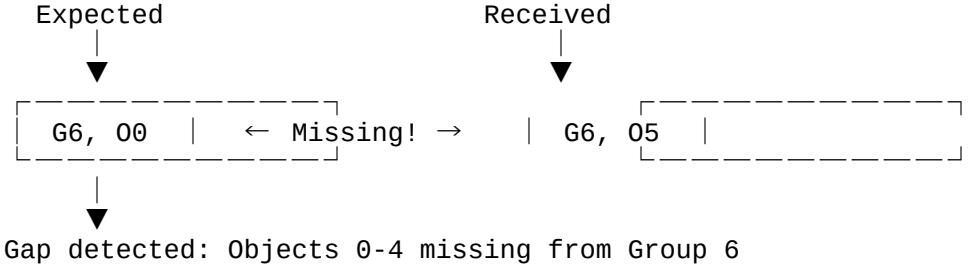
5.2.1. Detecting Missing Objects

Because objects within a group are sequentially numbered and groups themselves are ordered, subscribers can immediately identify when expected objects are missing by comparing received object identifiers against their expected sequence.

Gap Detection:

Subscriber State:
  Last received: Group 5, Object 999
  Expected next: Group 6, Object 0

Incoming object: Group 6, Object 5

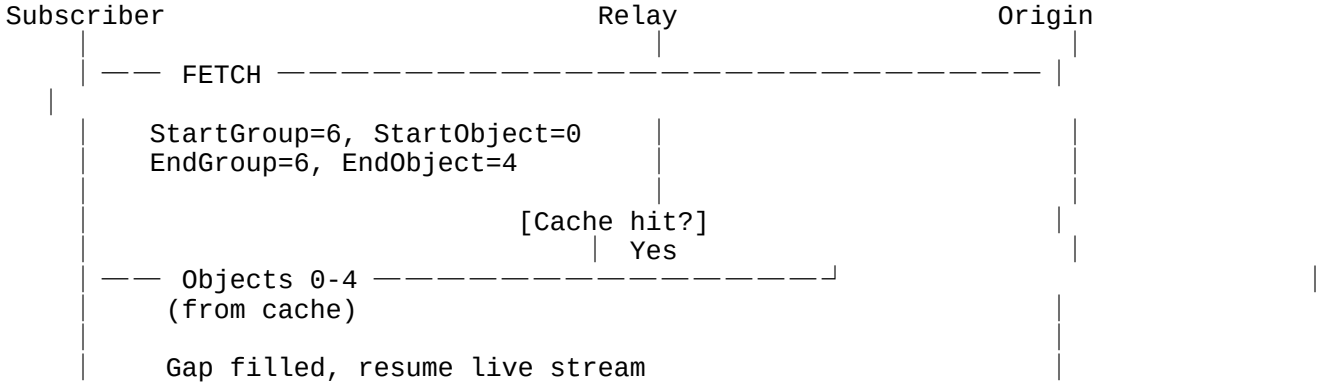


5.2.2. Recovery: Fetch Missing Objects

The preferred recovery approach is to fetch only the specific missing

objects using MOQT's FETCH operation with precise group and object boundaries. If relay caches contain the missing data, recovery completes without any load on the origin server.

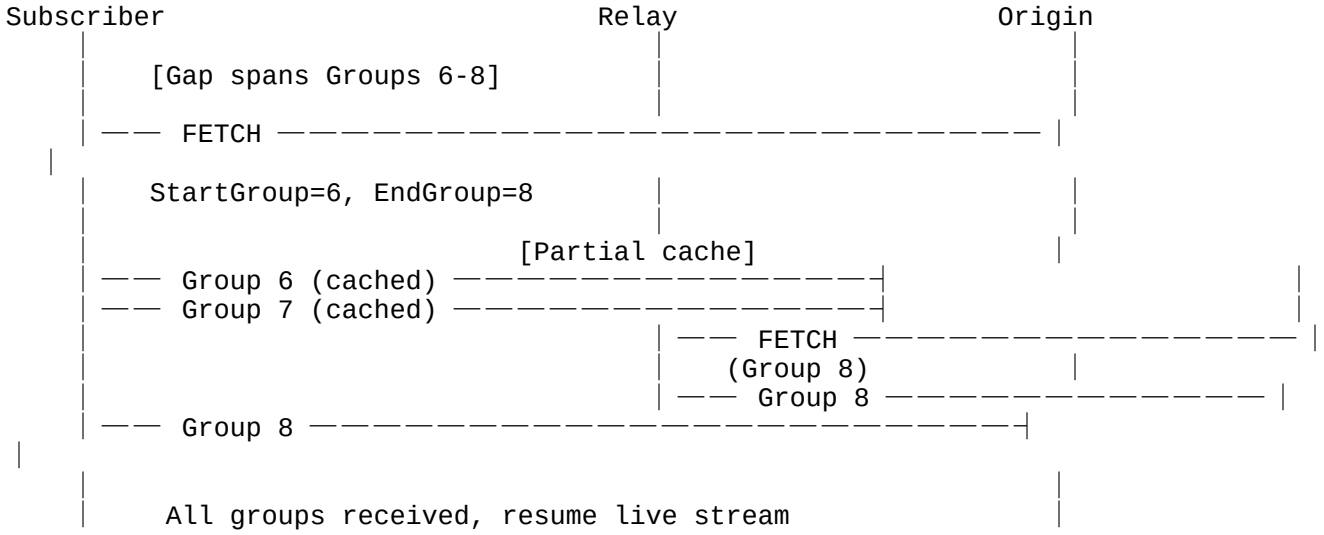
Fetch Missing Objects:



5.2.3. Recovery: Fetch Full Group

For larger gaps spanning multiple groups, subscribers may request entire groups rather than specifying individual object ranges. This approach is simpler but may retrieve more data than strictly necessary.

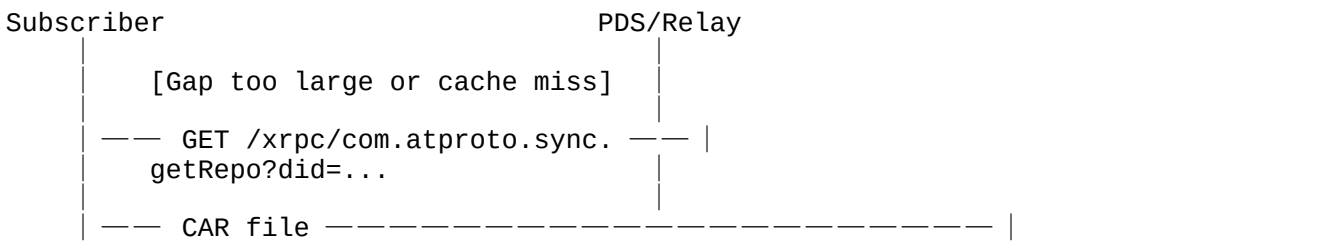
Fetch Full Group:



5.2.4. Recovery: Full Repository Sync

As a fallback when cached data is unavailable or gaps are too extensive, subscribers can perform a full repository synchronization using either HTTP CAR exports or the MOQT repository sync track. This approach validates the entire repository state against a #sync event.

Full Repository Sync:



```
|     (complete repository export)   |
|                                    |
|    [Validate against #sync event]  |
|                                    |
|    1. Parse CAR blocks             |
|    2. Verify MST root matches      |
|    3. Update local state           |
|    4. Resume live subscription     |
```

5.3. Disconnection Handling

MOQT over QUIC provides superior disconnection handling through three complementary mechanisms: 0-RTT session resumption, cursor-based reconnection, and transparent connection migration.

5.3.1. Brief Disconnections: 0-RTT Resumption

For brief disconnections (typically under 30 seconds), QUIC's 0-RTT resumption allows subscribers to restore their session with minimal latency. The subscriber presents a session ticket from the previous connection, and the relay resumes delivery from where it left off if the cached session state remains valid. This typically completes in a single round trip.

QUIC 0-RTT Resumption:

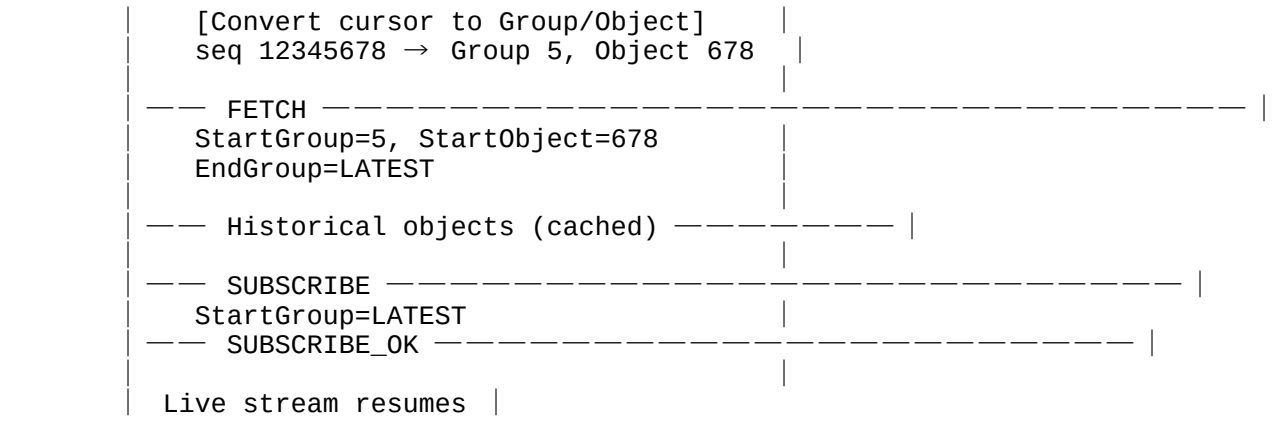
```
Subscriber                                  Relay
     |                                        |
     |    [Connection lost briefly]           |
     |                                        |
     |    [Reconnecting with session ticket]  |
     |                                        |
     | ——— QUIC 0-RTT ——————————————————————— |
     |    ClientHello + session ticket        |
     |    + early data (SUBSCRIBE)            |
     |                                        |
     | ——— ServerHello + Finished ——————————— |
     |                                        |
     | ——— Continue from last position ———————— |
     |     (relay cached subscriber state)    |
     |                                        |
     |  Resume live stream (minimal gap)      |
```

5.3.2. Extended Disconnections: Cursor-Based Reconnection

For extended disconnections where session state has expired, subscribers perform a full reconnection using their saved cursor position. The subscriber first uses FETCH to retrieve any missed events from their last known position, then establishes a new SUBSCRIBE starting from the latest group to receive live updates. Relay caches serve the historical data when available, minimizing load on upstream servers.

Full Reconnection with Cursor:

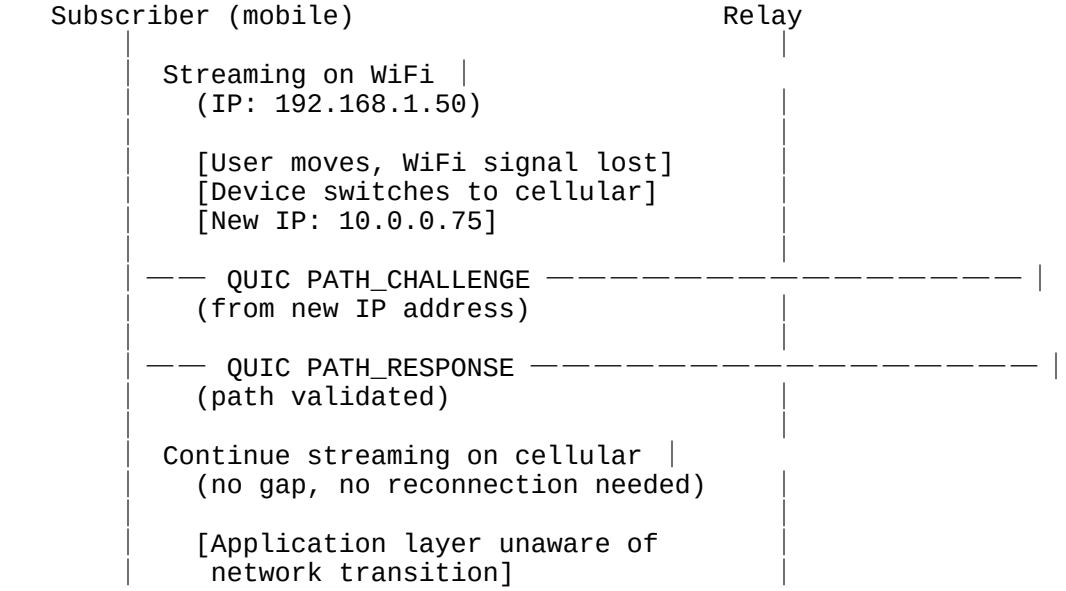
```
Subscriber                                  Relay
     |                                        |
     |    [Disconnected for extended period]  |
     |    [Saved state: at-seq = 12345678]    |
     |                                        |
     | ——— QUIC Handshake (full) ————————————— |
     | ——— Connection Established ———————————— |
     |                                        |
     | ——— CLIENT_SETUP ——————————————————————— |
     | ——— SERVER_SETUP ——————————————————————— |
     |                                        |
```



5.3.3. Network Transitions: Connection Migration

For network transitions such as WiFi-to-cellular handoffs, QUIC's connection migration enables seamless continuation without any application-layer intervention. The subscriber's IP address changes transparently while the MOQT session continues uninterrupted, ensuring no gaps in the event stream during mobile network transitions.

QUIC Connection Migration:



6. Authentication and Authorization

TODO

7. Security Considerations

TODO

8. IANA Considerations

TODO

9. References

9.1. Normative References

[AT-ARCH] Newbold, B. and D. Holmgren, "Authenticated Transfer (AT) Protocol Architecture", Work in Progress, Internet-Draft, draft-newbold-at-architecture, 2025,

<https://datatracker.ietf.org/doc/draft-newbold-at-architecture/>.

[AT-REPO] Holmgren, D. and B. Newbold, "Authenticated Transfer (AT) Repository and Synchronization", Work in Progress, Internet-Draft, draft-holmgren-at-repository, 2025, <https://datatracker.ietf.org/doc/draft-holmgren-at-repository/>.

[MoQ-TRANSPORT] Curley, L., Pugin, K., Nandakumar, S., Vasiliev, V., and I. Swett, "Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-transport-16, January 2026, <https://datatracker.ietf.org/doc/draft-ietf-moq-transport/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

9.2. Informative References

[MoQ-C4M] Jennings, S., "Common Access Token for Media over QUIC", Work in Progress, Internet-Draft, draft-ietf-moq-c4m, 2025, <https://datatracker.ietf.org/doc/draft-ietf-moq-c4m/>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/rfc/rfc7540>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[WebTransport] Vasiliev, V., "The WebTransport Protocol Framework", RFC 9297, August 2023, <https://www.rfc-editor.org/rfc/rfc9297>.

Authors' Addresses

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com

Cullen Jennings
Cisco Systems
Email: fluffy@cisco.com