Model Context Protocol
Internet-Draft
Intended status: Experimental
Expires: 23 April 2026

C. Jennings
Cisco Systems
I. Swett
Google
J. Rosenberg
Five9
S. Nandakumar
Cisco Systems

20 October 2025

Model Context Protocol over Media over QUIC Transport
draft-jennings-mcp-over-moqt-00

Abstract

This document defines how to use Media over QUIC Transport (MOQT) as the underlying transport protocol for the Model Context Protocol (MCP). MCP is a protocol that enables seamless integration between language model applications and external data sources and tools. MOQT provides efficient, low-latency, publish-subscribe media delivery over QUIC and WebTransport. This specification describes the mapping of MCP messages onto MOQT objects and defines the procedures for establishing and maintaining MCP sessions over MOQT.

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://example.org/mcp-moqt/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-mcp-over-moqt/.

Discussion of this document takes place on the Model Context Protocol Working Group mailing list (mailto:mcp@example.org), which is archived at https://example.org/mcp/.

Source for this draft and an issue tracker can be found at https://github.com/example/mcp-moqt.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

This Internet-Draft will expire on 23 April 2026.

Copyright Notice

Table of Contents

1. Introduction

The Model Context Protocol (MCP) [MCP] is an open protocol that enables seamless integration between LLM applications and external data sources and tools. MCP uses JSON-RPC 2.0 for message exchange and can operate over various transport protocols.

Media over QUIC Transport (MOQT) [MOQT] is a protocol optimized for QUIC [QUIC] that provides efficient publish-subscribe media delivery with support for content distribution networks and low-latency requirements.

This document outlines the use of MOQT as the transport layer for MCP, utilizing MOQT's object delivery and QUIC's multiplexed streams for low-latency communication. Paired unidirectional tracks enable real-time interactions between language models and external systems. MOQT's object-based delivery allows efficient resource and tool sharing across clients without redundant transfers. Relay networks support scalable content distribution, enabling global MCP deployment with optimized performance. Native prioritization ensures critical operations like tool execution and user interactions receive sufficient bandwidth. QUIC's connection migration and recovery provide reliable connectivity in mobile and unstable networks.

1.1. Motivation

The Model Context Protocol (MCP) enables AI applications to integrate with external data sources and tools, but current transport mechanisms
poses challenges that can hinder optimal performance and scalability. This specification addresses these challenges by leveraging Media over QUIC Transport (MOQT) as an alternate transport layer specifically optimized for MCP's operational requirements.

1.1.1. Current Transport Challenges

WebSocket Transport lacks native prioritization mechanisms, causing critical control messages and urgent tool operations to compete equally with routine resource updates. Head-of-line blocking occurs when large resource transfers delay time-sensitive operations, and the protocol provides no built-in caching or relay capabilities for scalable content distribution.

HTTP Transport operates on a strict request-response model that conflicts with MCP's inherently event-driven architecture. This mismatch forces artificial polling patterns for resource updates and prevents efficient server-initiated notifications. While Server Side Streaming provides a mechanism for servers to send asynchronous

events, it introduces significant challenges including connection complexity, limited browser support, proxy interference, and difficultly in handling bidirectional communication patterns essential for interactive MCP operations. Additionally, HTTP/1.1's connection limitations and HTTP/2's stream dependencies create bottlenecks in high-throughput AI applications.

1.1.2. MCP over MOQT

MOQT's publish-subscribe architecture provides an ideal foundation for MCP operations, where MCP servers naturally function as publishers of resources, tools, and capabilities, while clients subscribe to relevant content streams. This alignment enables several key architectural benefits:

Semantic Mapping: MCP's resource-centric model maps directly to MOQT's track-based content organization. Each resource, tool, or capability becomes a dedicated track with independent lifecycle management, version control, and access policies.

Event-Driven Communication: MOQT's native support for asynchronous object delivery matches MCP's notification-based architecture, eliminating the artificial request-response constraints imposed by HTTP transport.

Priority-Aware Delivery: MOQT's built-in prioritization system enables sophisticated quality-of-service policies, ensuring that critical MCP operations like tool execution and user interactions receive appropriate bandwidth allocation and reduced latency.

1.1.3. Operational Advantages

The combination of MCP and MOQT can offer operational improvements along the following dimensions:

Low Latency Operations: MOQT's object-based delivery model eliminates traditional HTTP request-response overhead, enabling direct message passing with minimal protocol overhead. Tool execution requests are sent as prioritized FETCH operations while control messages receive the highest priority (1-2), ensuring responsive interaction even under high system load.

Bandwidth Efficiency: MOQT relays provide sophisticated caching mechanisms and subscription aggregations, that deduplicate frequently accessed resources across multiple clients. Large documentation sets, configuration files, and static assets are cached at relay points, dramatically reducing origin server load while enabling selective updates through version-aware object delivery.

Multiplexing and Concurrency: QUIC’s multiplexing capabilities prevent head-of-line blocking between different MCP operations, allowing hundreds of concurrent tool executions, resource subscriptions, and notification streams to operate independently without mutual interference.

Network Resilience and Mobility: QUIC’s connection migration and 0-RTT reconnection capabilities maintain session continuity across network changes, a critical requirement for mobile AI applications and long-running workflows that may experience network interruptions.

Edge-Optimized Content Distribution: The relay infrastructure supports geographically distributed caching and intelligent content placement, enabling global MCP deployments with optimized performance characteristics tailored to regional access patterns and data locality requirements.

1.2. Terminology

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.

This document uses terminology from [MCP] and [MOQT].

1.3. Protocol Overview

MCP uses MOQT to map its abstractions to MOQT’s transport mechanisms. MCP messages, encoded as JSON-RPC 2.0, are embedded in MOQT payloads, ensuring compatibility with MCP formats and efficient transport over QUIC streams.

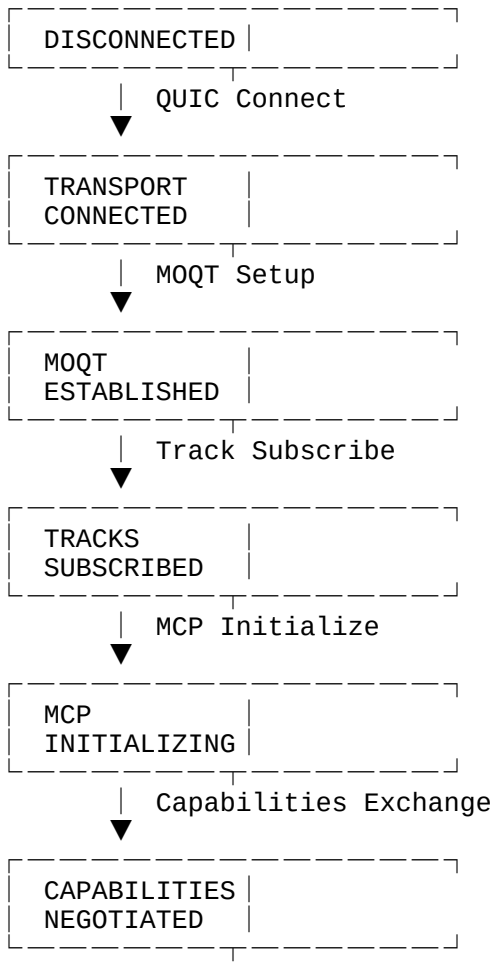
MCP sessions use MOQT sessions for communication, starting with a handshake to negotiate capabilities. This phase ensures clients and servers agree on features, resource types, and parameters.

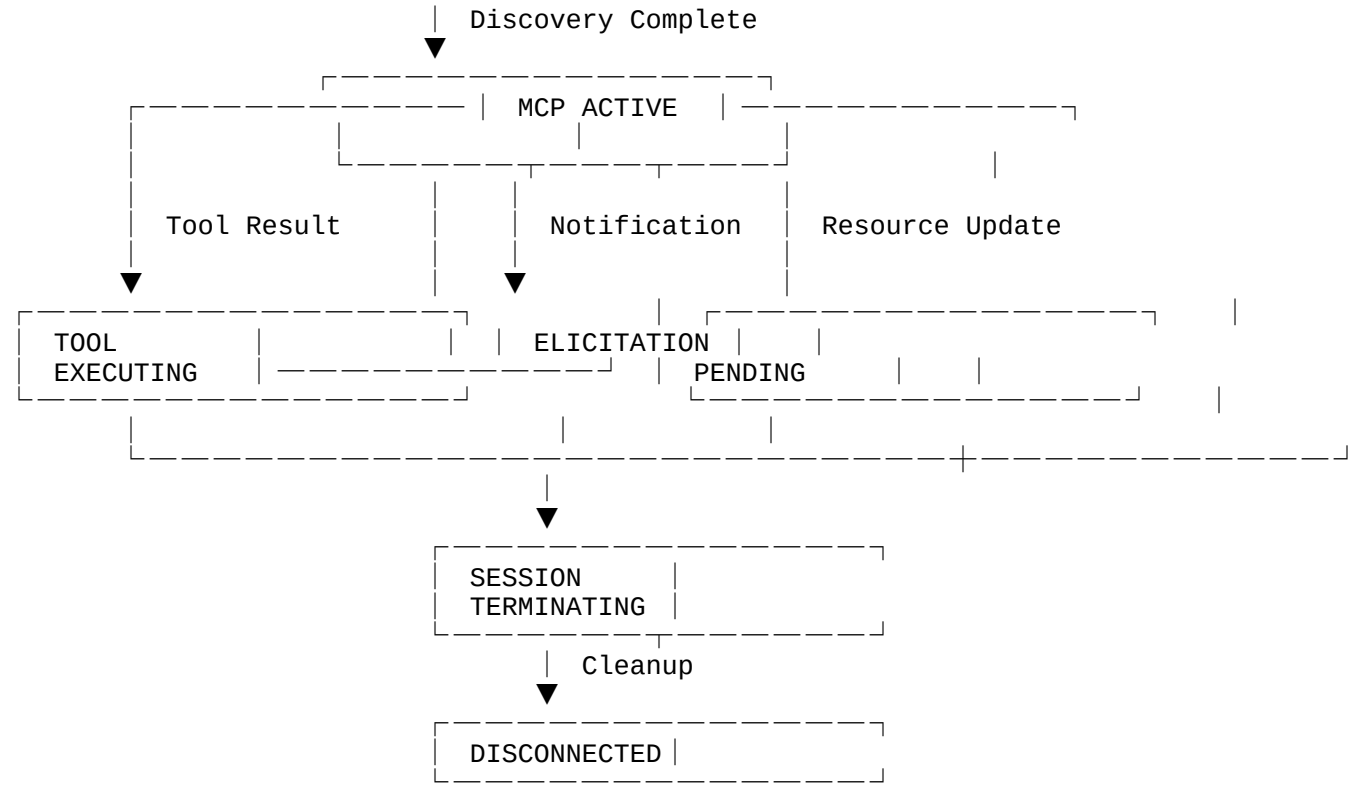
MCP Resources are server-controlled data sources exposed to clients for read access. In MOQT, each resource is published over *resources* track. Content is delivered via MOQT objects, supporting various data types like files, API responses, database results, and version control history. Streaming enables efficient handling of large resources, while automatic updates use new objects with updated sequence numbers for incremental changes. This supports advanced caching and efficient client synchronization.

Tools enable servers to execute client-requested operations. MOQT maps tools to dedicated *tools* tracks and uses FETCH for request-response patterns. Supported operations include API calls, file tasks, computations, and service integrations. Progress tracking uses notifications for real-time updates. FETCH ensures robust execution with prioritization, multiplexing, error handling, and timeout management.

Notifications about resource changes and events are published in specialized tracks, allowing clients to subscribe without affecting other channels. User input is managed through interactive control tracks for consent and data collection.

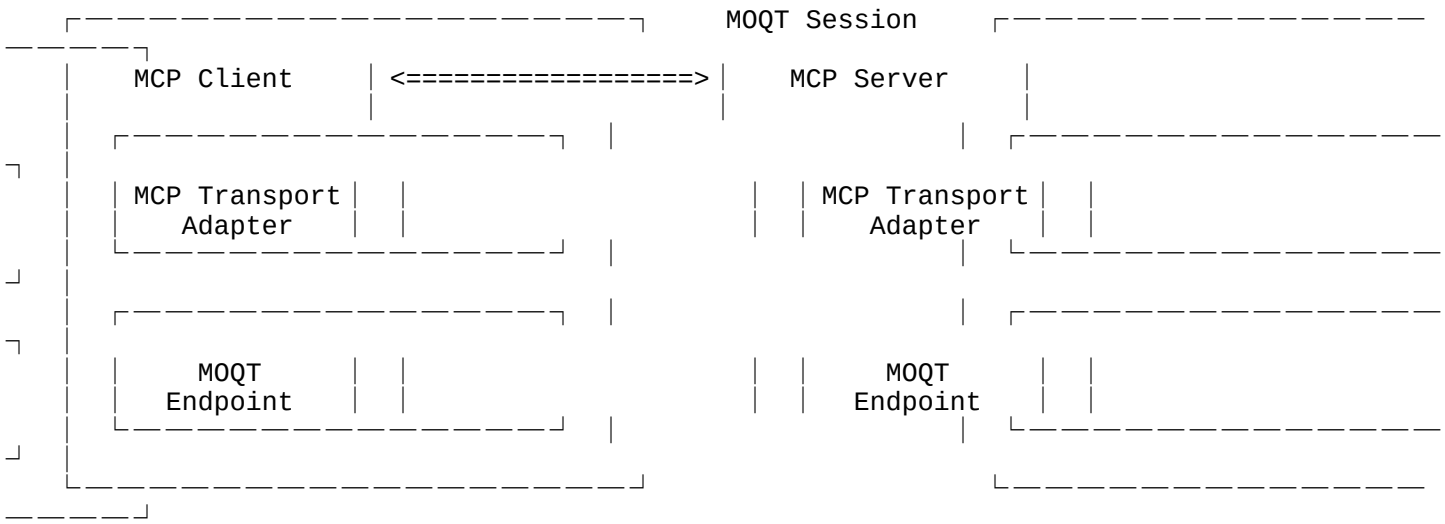
1.3.1. MCP Lifecycle State Chart





1.3.2. System Components

Below figure shows system components proposed in this specification.



The MCP client-host-server architecture maps to MOQT as follows:

* *MCP Host*: Acts as MOQT client, managing multiple server connections
* *MCP Client*: Implemented as MCP transport adapter within the host
* *MCP Server*: Acts as MOQT publisher, exposing resources, tools, and prompts

2. MOQT Tracks for MCP

2.1. Control Flow

The control flow establishes the fundamental communication channels for MCP session management and coordination. This specifcation prposes 2 unidirectional tracks for direction for performing session establishment, capability negotiation, and various control operations using the MOQT’s publish-subscribe semantics.

2.1.1. Client-to-Server Control Track

* *Publisher*: MCP Client
* *Subscribers*: MCP Server, optional monitoring relays
* *Content*:
  - MCP initialize requests
  - Ping messages (client-initiated)
  - Capability negotiation requests
  - Session teardown requests
  - Tool cancellation requests

2.1.2. Server-to-Client Control Track

* *Publisher*: MCP Server
* *Subscribers*: MCP Client, optional monitoring relays
* *Content*:
  - MCP initialize responses
  - Pong messages (server responses)

- Capability negotiation responses
- Session teardown confirmations
- Server-initiated status updates

2.2. Track Types

The various sub-sections provides information on track naming design defined by this specification to map to MCP operations.

2.2.1. Control Tracks

Control tracks map MCP session management and coordination messages including initialization, ping/pong heartbeats, and capability negotiation and 2 unidirectional tracks for server and client to carryout message exchanges.

Track namespace:

(mcp, <session-id>, control)

Track name (Client-to-Server):

(client-to-server)

Track name (Server-to-Client):

(server-to-client)

Groups and objects are mapped as follows:

Each MCP control message is encoded as a single MOQT object within a group. Group IDs are assigned sequentially starting from 0, incrementing for each new control message. Object IDs within each group are always 0 since each control message maps to exactly one object. Objects contain JSON-encoded MCP control messages as defined in the MCP specification.

2.2.2. Resource Tracks

Resource tracks map MCP resources, delivering server-published content and metadata that serve as the main data sources exposed by MCP servers to clients.

Track namespace:

(mcp, <session-id>, resources)

Track name:

(<resource-uri>)

Groups and objects are mapped as follows:

Each resource version is assigned a unique group ID, starting from 0 and incrementing with each resource update. Within each group, objects represent chunks or segments of the resource content. Object IDs start at 0 and increment sequentially for each chunk. Objects contain the actual resource data encoded according to the resource's declared content type (binary data, JSON, text, etc.).

2.2.3. Tool Tracks

Tool tracks map MCP tool execution, facilitating tool invocation requests, responses, and progress updates using MOQT's FETCH operations.

Track namespace:

(mcp, <session-id&>, tools)

Track name:

(<tool-name>)

Groups and objects are mapped as follows:

Each tool invocation creates a new group with a unique group ID assigned by the requesting client. Within each group, object ID 0 contains the tool request (JSON-encoded MCP tool call), subsequent object IDs contain tool responses and progress updates. Objects are JSON-encoded according to the MCP tool execution protocol.

2.2.4. Prompt Tracks

Prompt tracks map MCP prompts, distributing pre-defined templates and instructions that standardize common operations across the MCP ecosystem.

Track namespace:

(mcp, <session-id>, prompts)

Track name:

(<prompt-name>)

Groups and objects are mapped as follows:

Each prompt version is assigned a unique group ID starting from 0, incrementing with each prompt update. Within each group, object ID 0 contains the prompt template and metadata. Objects contain JSON-encoded prompt definitions including template text, parameter schemas, and versioning information.

2.2.5. Notification Tracks

Notification tracks map MCP notifications, providing asynchronous event delivery for server-sent notifications, progress updates, and system events.

Track namespace:

(mcp, <session-id>, notifications)

Track name:

(<category>)

The category parameter classifies notification types to enable efficient subscription management and routing. Common category values include:

* progress - Tool execution progress updates and status reports
* resources - Resource change notifications (listChanged, updated)
* prompts - Prompt template updates and availability changes
* system - Server status updates, connection events, error conditions
* elicitation - User input request notifications and responses
* tools - Tool availability changes and capability updates

Custom categories may be defined for application-specific notification types following the pattern <domain>/<type> (e.g., ai/model_updated, workspace/file_changed).

Groups and objects are mapped as follows:

Each notification event creates a new group with group IDs assigned sequentially starting from 0. Within each group, object ID 0 contains the notification payload. Objects contain JSON-encoded notification messages as defined in the MCP specification, including event type, timestamp, and payload data.

2.2.6. Elicitation Tracks

Elicitation tracks map MCP user input collection, handling interactive flows for gathering user consent and data through pairs of unidirectional tracks. The elicitation flow uses a unidirectional track pair to collect user input efficiently while preserving privacy. Each request creates isolated track pairs, allowing concurrent operations. Servers request user consent and input, while clients retain control to reject or modify requests.

Track namespace:

(mcp, <session-id>, elicitation, <request-id>)

Track name (Server-to-Client):

(<server-to-client>)

Track name (Client-to-Server):

(<client-to-server>)

The Server-to-Client elicitation track is published by the MCP Server and subscribed to by the MCP Client, carrying:

* Elicitation request initiation
* Input schema definitions
* Validation error messages
* Request timeout notifications
* Request cancellation notices

The Client-to-Server elicitation track is published by the MCP Client and subscribed to by the MCP Server, carrying:

* User consent responses
* Input data submissions

* Request rejection notifications

* Request cancellation confirmations

Groups and objects are mapped as follows:

Each elicitation exchange creates new groups with group IDs starting from 0 and incrementing for each message in the exchange. Within each group, object ID 0 contains the elicitation message (request, response, or status update). Objects contain JSON-encoded elicitation messages including input schemas, validation requirements, and user responses.

2.2.7. Log Tracks

Log tracks map MCP logging information, providing debugging and monitoring capabilities by carrying diagnostic information for system troubleshooting and performance optimization.

Track namespace:

(mcp, <session-id>, logs)

Track name:

(<category>)

The category parameter organizes log entries by severity level and source component to enable selective monitoring and efficient log processing. Standard category values include:

* error - Error conditions, exceptions, and critical failures

* warn - Warning messages and non-critical issues requiring attention

* info - General informational messages about system operations

* debug - Detailed debugging information for troubleshooting

* trace - Fine-grained execution tracing for performance analysis

* audit - Security and compliance audit trails

* metrics - Performance metrics and system statistics

Component-specific categories may be defined using the pattern <level>/<component> (e.g., error/transport, debug/tools, info/resources) to provide granular filtering capabilities.

Groups and objects are mapped as follows:

Each log entry creates a new group with group IDs assigned sequentially starting from 0, ordered by log timestamp. Within each group, object ID 0 contains the log entry. Objects contain JSON-encoded log messages including severity level, timestamp, source component, and message content.

3. Protocol Operation

3.1. MOQT Session Establishment

The session establishment process begins with establishing the underlying MOQT session, which can be either a direct QUIC connection or a WebTransport [WebTransport] session depending on the deployment environment. QUIC connections provide optimal performance for server-to- server communication, while WebTransport enables browser-based clients to participate in MCP sessions.

The MOQT handshake phase involves the exchange of CLIENT_SETUP (0x20) and SERVER_SETUP (0x21) messages that negotiate protocol versions, supported features, and operational parameters.

3.2. Proposed Session ID Discovery

MCP sessions require unique identifiers to organize track namespaces and enable proper message routing. This section defines a well-known track approach for session discovery that allows clients to dynamically discover available MCP services and obtain session identifiers through standardized FETCH operations.

3.2.1. Well-Known Discovery Track

The discovery mechanism uses a well-known track namespace and name that all MCP servers must support. This track serves as the entry point for clients to discover available MCP sessions and obtain the necessary session identifiers for subsequent operations.

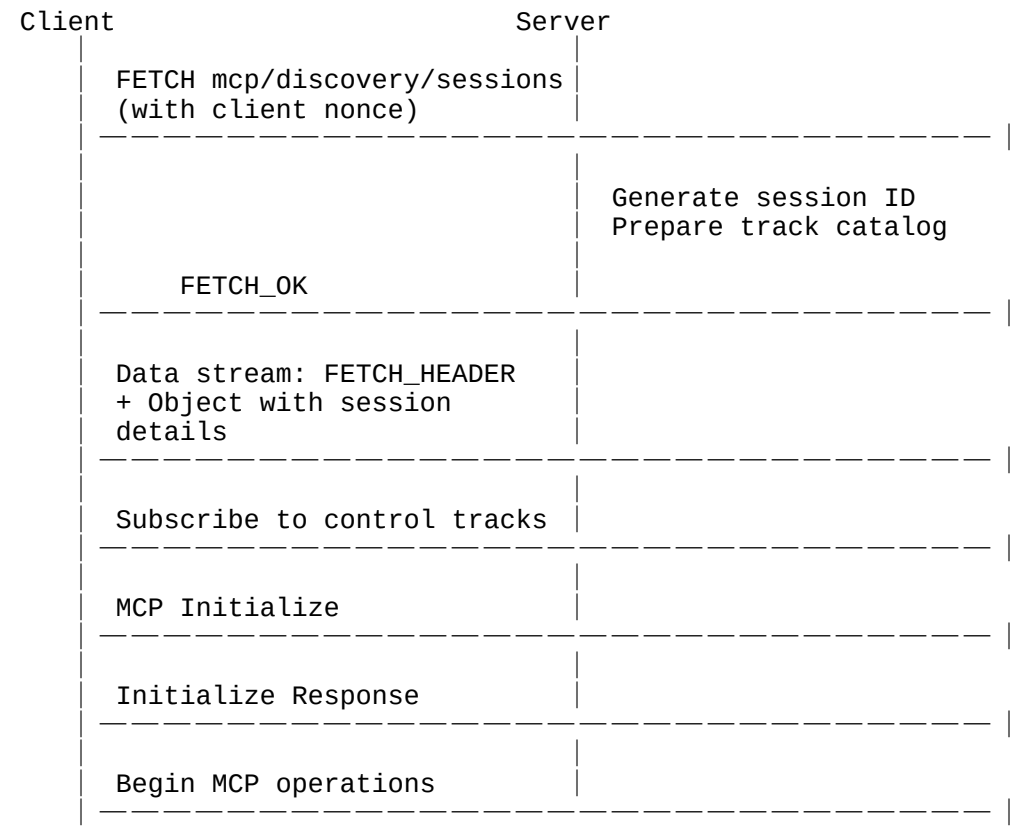
Track Namespace: mcp/discovery
Track Name: sessions

Clients initiate discovery by sending a FETCH request to this well-known track, providing a client-side nonce or identifier as a parameter. The server responds with a JSON-encoded MOQT object containing server details, a newly minted session identifier, and information about available tracks on that server.

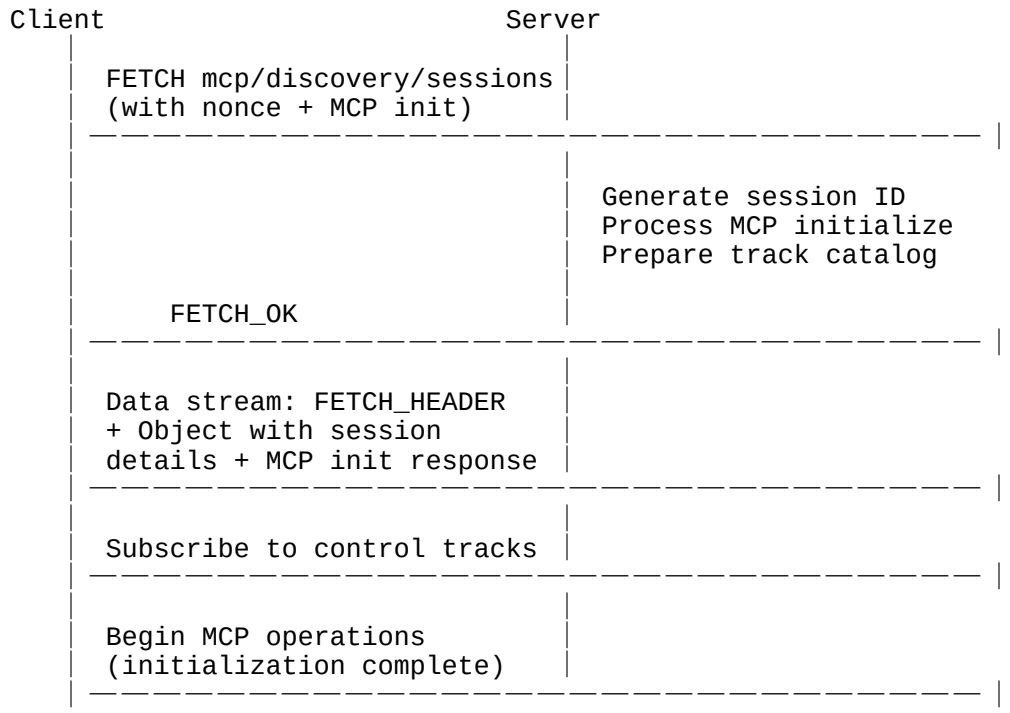
3.2.2. Discovery Flow

The session discovery process follows this sequence:

3.2.2.1. Standard Discovery Flow



3.2.2.2. Optimized Discovery Flow (RTT Reduction)



3.2.3. Discovery Request Format

Clients send FETCH requests to the well-known discovery track with a client-generated nonce or identifier in the payload:

```
FETCH {
  Type (i) = 0x16,
  Length (16),
  Request ID (i) = 1,
  Subscriber Priority (8) = 30,
  Group Order (8) = 0,
  Fetch Type (i) = 0x1,  // Standalone
  Track Namespace (tuple) = (... "mcp", "discovery"),
  Track Name Length (i) = 8,
  Track Name = "sessions",
  Start Location = {Group: 0, Object: 0},
  End Location = {Group: 0, Object: 1},
  Number of Parameters (i) = 1,
  Parameters = {
    "mcp_payload": {
      "jsonrpc": "2.0",
      "id": 1,
      "method": "discovery/request_session",
      "params": {
        "client_nonce": "client-nonce-abc123",
        "client_info": {
          "name": "ExampleClient",
          "version": "1.0.0"
        },
        "requested_capabilities": ["resources", "tools", "prompts"]
      }
    }
  }
}
```

3.2.3.1. RTT Optimization with Combined Initialize

To reduce round-trip time, clients MAY include their MCP initialize request within the discovery FETCH payload. This allows the server to perform both session discovery and MCP initialization in a single exchange:

```
FETCH {
  Type (i) = 0x16,
  Length (16),
  Request ID (i) = 1,
  Subscriber Priority (8) = 30,
  Group Order (8) = 0,
  Fetch Type (i) = 0x1,  // Standalone
  Track Namespace (tuple) = ("mcp", "discovery"),
  Track Name Length (i) = 8,
  Track Name = "sessions",
  Start Location = {Group: 0, Object: 0},
  End Location = {Group: 0, Object: 1},
  Number of Parameters (i) = 1,
  Parameters = {
    "mcp_payload": {
      "jsonrpc": "2.0",
      "id": 1,
      "method": "discovery/request_session_with_init",
      "params": {
        "client_nonce": "client-nonce-abc123",
        "client_info": {
          "name": "ExampleClient",
          "version": "1.0.0"
        },
        "requested_capabilities": ["resources", "tools", "prompts"],
        "mcp_initialize": {
          "protocolVersion": "2025-06-18",
          "capabilities": {
            "resources": { "subscribe": true },
            "tools": { "progress": true },
            "prompts": {},
            "elicitation": {}
          },
          "clientInfo": {
            "name": "ExampleClient",
            "version": "1.0.0"
          }
        }
      }
    }
  }
}
```

3.2.4. Discovery Response Format

The server responds with a JSON-encoded MOQT object containing the newly minted session ID and available track information:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "session_id": "session-uuid-456",
    "server_info": {
      "name": "ExampleServer",
      "version": "2.0.0",
      "protocol_version": "2025-06-18"
    },
    "available_tracks": {
      "control": {
        "client_to_server":
           "mcp/session-uuid-456/control/client-to-server",
        "server_to_client":
           "mcp/session-uuid-456/control/server-to-client"
      },
      "resources": [
        {
          "name": "documentation",
          "track": "mcp/session-uuid-456/resources/documentation",
          "content_type": "text/markdown",
          "description": "API documentation and guides"
        }
      ],
      "tools": [
        {
          "name": "file_operations",
          "track": "mcp/session-uuid-456/tools/file_operations",
          "description": "File system operations"
        }
      ],
      "prompts": [
        {
          "name": "code_review",
          "track": "mcp/session-uuid-456/prompts/code_review",
          "description": "Code review prompt templates"
        }
      ]
    },
    "session_expires": "2025-06-18T12:00:00Z"
  }
}
```

3.2.4.1. Combined Discovery and Initialize Response

When the client uses the RTT optimization by including MCP initialize in the discovery request, the server responds with both session discovery and MCP initialization results:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "session_id": "session-uuid-456",
    "server_info": {
      "name": "ExampleServer",
      "version": "2.0.0",
      "protocol_version": "2025-06-18"
    },
    "available_tracks": {
      "control": {
        "client_to_server":
           "mcp/session-uuid-456/control/client-to-server",
        "server_to_client":
           "mcp/session-uuid-456/control/server-to-client"
      },
      "resources": [
        {
          "name": "documentation",
          "track": "mcp/session-uuid-456/resources/documentation",
          "content_type": "text/markdown",
          "description": "API documentation and guides"
        }
      ],
      "tools": [
        {
          "name": "file_operations",
          "track": "mcp/session-uuid-456/tools/file_operations",
          "description": "File system operations"
        }
      ],
      "prompts": [
        {
          "name": "code_review",
          "track": "mcp/session-uuid-456/prompts/code_review",
          "description": "Code review prompt templates"
        }
      ]
    },
    "session_expires": "2025-06-18T12:00:00Z",
    "mcp_initialize_response": {
```

```
        "protocolVersion": "2025-06-18",
        "capabilities": {
          "resources": { "subscribe": true, "listChanged": true },
          "tools": { "progress": true },
          "prompts": { "listChanged": true },
          "elicitation": {}
        },
        "serverInfo": {
          "name": "ExampleServer",
          "version": "2.0.0"
        }
      }
    }
  }
```

3.2.5. Session Establishment Flow

After receiving the discovery response, clients proceed with MCP session establishment using the provided session ID. The flow depends on whether the RTT optimization was used:

3.2.5.1. Standard Discovery Flow

1. Track Subscription: Client subscribes to the control tracks using the session ID from the discovery response

2. MCP Initialize: Client sends the standard MCP initialize message on the client-to-server control track

3. Capability Negotiation: Server responds with its capabilities on the server-to-client control track

4. Session Active: Client begins normal MCP operations using the discovered tracks and session ID

3.2.5.2. Optimized Discovery Flow (with combined initialize)

1. Track Subscription: Client subscribes to the control tracks using the session ID from the discovery response

2. Session Active: Since MCP initialization was completed during discovery, the client can immediately begin normal MCP operations using the discovered tracks and negotiated capabilities

The RTT optimization reduces the session establishment from 4 round-trips to 2 round-trips by combining discovery and initialization operations.

3.3. Priority Management

MOQT's object priorities optimize MCP message delivery by considering the importance of different track types specified.

The following priority assignments are RECOMMENDED, but applications may adjust them as needed.

1. Control Messages: Highest priority (1-10)

2. Tool Operations: High priority (20-30)

3. Notifications: Medium priority (40-60)

4. Resource Operations: Lower priority (70-90)

5. Logs: Lowest priority (100+)

Priorities can also be adjusted based on:

* User interaction urgency (elicitation requests get higher priority)

* Tool execution criticality (error handling gets priority boost)

* Resource size and frequency (large resources get lower priority)

* Server load conditions (adaptive priority scaling)

3.4. Error Handling

TODO

4. Relay Support

MOQT relays enable scalable MCP deployments by providing intelligent aggregation, caching, and distribution capabilities that optimize performance for AI workflows and reduce server load across multiple clients.

4.1. Subscription Aggregation

MOQT relays provide significant efficiency gains by aggregating multiple client subscriptions to the same content into single upstream requests. When multiple clients subscribe to identical MCP resources, tools metadata, or notification streams, the relay consolidates these into a single subscription to the origin server, dramatically reducing server load and network bandwidth consumption.

4.1.1. Resource Subscription Aggregation

When multiple clients subscribe to the same resource track, the relay maintains a single upstream subscription and fans out content to all interested clients:

```
Client A subscribes: mcp/{session-id}/resources/documentation
Client B subscribes: mcp/{session-id}/resources/documentation
Client C subscribes: mcp/{session-id}/resources/documentation

Relay behavior:
- Single upstream subscription: mcp/{session-id}/resources/documentation
- Fan-out to 3 downstream clients
- Bandwidth savings: 3x reduction in server load
```

The relay tracks subscription interest levels and automatically establishes upstream subscriptions when the first client subscribes to a track, and tears down the upstream subscription when the last client unsubscribes.

4.1.2. Tool Metadata Aggregation

Tool discovery operations benefit significantly from aggregation since multiple clients typically need access to the same tool schemas and capabilities

TODO: Add an examples

4.2. Content Caching for MCP and AI Workflows

4.2.1. Static Resources and Documentation

Large documentation sets, API references, and knowledge bases represent the most cache-friendly content in MCP deployments. These resources are accessed frequently across multiple AI sessions but change infrequently, making them ideal candidates for aggressive caching.

Static resources are delivered through dedicated tracks like mcp/{session-id}/resources/{resource-uri} and benefit significantly from relay caching since multiple clients often access the same content.

4.2.2. Tool Schema and Configuration Caching

Tool schemas, parameter definitions, and configuration metadata are cached aggressively since they define the stable interface contracts between AI applications and external services. This metadata is cached with high TTL values, allowing relays to serve tool discovery requests without repeatedly querying the origin server. Schema definitions are published as MOQT objects that remain valid until tools are modified or removed, making them ideal candidates for aggressive aching policies. In case of modifications, new updated schema definitions are published with updated versioning info,

4.2.3. Capability Information and Session Metadata

Server capability announcements that define supported features, resource types, and operational parameters negotiated during session establishment are cached via mcp/{session-id}/control/capabilities tracks as one-time objects published after MCP initialization, enabling relays to quickly respond to capability queries without server involvement. The caching strategy uses long-term storage with explicit invalidation only on server restart or capability changes.

Session metadata comprises initialization parameters, connection preferences, and configuration data that define session characteristics and relay behavior. This metadata is distributed through mcp/{session-id}/control/metadata tracks as cacheable objects, with relay configuration parameters embedded in SERVER_SETUP message extensions. The cached metadata enables consistent session behavior across client reconnections and provides relays with the information needed for optimal routing and caching decisions.

4.2.4. Model Context and History Caching

For AI workflows that involve multi-turn conversations or context building, relays can cache conversation contexts and interaction histories to support session resumption and context sharing:

4.3. Track Name Discovery and Dynamic Updates

MOQT relays enhance the MCP experience by providing efficient service-level discovery mechanisms that help clients understand available resources, tools, and services within established MCP sessions without requiring prior configuration knowledge.

4.3.1. Discovery Track Implementation

Relays implement service-level discovery tracks that catalog available MCP services and their associated track patterns within a specific session:

Track namespace: mcp/<session-id>/discovery Track name: catalog

```
{
  "server_catalog": {
    "mcp/{session-id}/resources": {
      "available_tracks": [
        "documentation", "api_schemas", "examples", "templates"
      ],
      "content_types": ["text/markdown",
             "application/json", "text/plain"],
      "update_frequency": "daily",
      "cache_recommended": true
    },
    "mcp/{session-id}/tools/": {
      "available_tracks": [
        "file_operations", "database_query", "code_analysis"
      ],
      "execution_types": ["synchronous", "asynchronous"],
      "progress_tracking": true,
      "cache_recommended": false
    },
    "mcp/{session-id}/prompts/": {
      "available_tracks": [
        "code_review", "data_analysis", "content_generation"
      ],
      "template_versions": ["1.0", "1.1", "2.0"],
      "cache_recommended": true
    }
  },
  "discovery_timestamp": "2025-06-18T10:30:00Z"
}
```

4.3.2. Client Discovery via FETCH Operations

Clients can efficiently discover available services by issuing FETCH requests to discovery tracks, receiving comprehensive information about available MCP capabilities

4.3.3. Asynchronous Updates via Subscriptions

Beyond one-time discovery, clients can subscribe to discovery tracks to receive real-time updates when new services become available or existing services change their characteristics

This subscription-based discovery update mechanism ensures that AI applications can dynamically adapt to changing service availability without requiring configuration changes or manual intervention.

5. Agent Skills and MCP Integration

Agent Skills represent a powerful extension to the MCP ecosystem, providing modular capabilities that can be efficiently delivered and managed through MOQT's transport mechanisms. This section describes how Skills integrate with MCP over MOQT.

5.1. Skills Architecture Overview

Agent Skills are filesystem-based resources that extend AI capabilities through three progressive loading levels, along the following dimensions:

* Always loading metadata Loading (~100 tokens) provides basic skill identification and capabilities

* Dynamicaly loading instructionns when triggered (under 5k tokens) containing skill logic and execution parameters

* Resources and Code are loaded as needed for actual skill execution and data processing

This progressive disclosure model aligns naturally with MOQT's object-based delivery system, where each loading level can be mapped to separate MOQT objects or tracks for optimal bandwidth utilization.

5.2. Skills Transport over MOQT

Skills metadata is distributed through dedicated discovery tracks that enable clients to understand available capabilities without loading full skill implementations:

Track namespace: mcp/<session-id>/skills Track name: catalog

The skills catalog provides comprehensive metadata about available skills, their capabilities, dependencies, and loading requirements:

```
{
  "available_skills": [
    {
      "skill_id": "powerpoint-processor",
      "name": "PowerPoint Processing",
      "description": "Create and modify PowerPoint presentations",
      "metadata_size": 95,
      "instructions_size": 4200,
      "resource_dependencies": ["office-templates", "style-schemas"],
      "security_level": "trusted",
      "loading_priority": "on-demand"
    },
    {
      "skill_id": "pdf-analyzer",
      "name": "PDF Analysis",
      "description": "Extract and analyze content from PDF documents",
      "metadata_size": 87,
      "instructions_size": 3800,
      "resource_dependencies": ["pdf-schemas"],
      "security_level": "sandboxed",
      "loading_priority": "preload"
    }
  ],
  "catalog_version": "1.2.0",
  "last_updated": "2025-06-18T10:30:00Z"
}
```

Skills leverage MOQT's object-based delivery for efficient progressive loading:

Each skill's metadata is delivered as a single MOQT object containing basic capability information, version data, and loading requirements.

Track namespace: 'mcp/<session-id>/skills/<skill-id>'
Track name: 'metadata'

When a skill is activated, its instruction set is loaded through dedicated objects containing execution logic, parameter schemas, and operational guidelines.

Track namespace: 'mcp/<session-id>/skills/<skill-id>'
Track name: 'instructions'

Skill resources such as templates, schemas, and code modules are loaded on-demand through separate resource tracks, enabling fine-grained loading control.

Track namespace: ‘mcp/<session-id>/skills/<skill-id>‘
Track name: ‘resources/<resource-type>‘

5.3. Skills Composition and Workflow Integration

Complex workflows can compose multiple skills together, with MOQT providing efficient coordination:

* Dependency Resolution: Skills declare dependencies that are automatically resolved through catalog metadata

* Execution Orchestration: Workflow engines can preload skill sets based on anticipated execution patterns

* Resource Sharing: Common skill resources are shared across workflow steps to minimize loading overhead

Skills can share context and state through dedicated context tracks:

Track namespace: mcp/<session-id>/skills/context Track name: <workflow-id>

This enables sophisticated multi-step workflows where skills build upon each other’s outputs while maintaining clean separation of concerns.

MOQT relays implement sophisticated caching for skill distribution:

* Metadata Caching: Skill catalogs cached with high TTL for quick discovery

* Instruction Caching: Frequently used skills cached at edge locations

* Resource Deduplication: Common skill resources shared across multiple skills

* Version Management: Skill updates distributed efficiently through version-aware object delivery

AI applications can implement predictive skill loading based on user patterns and workflow analysis, using MOQT’s subscription management to preload likely-needed skills while avoiding bandwidth waste.

6. Shared Context Between One MCP Client and Multiple MCP Servers

When a single MCP client connects to multiple MCP servers simultaneously, shared context management becomes critical for maintaining consistency, efficiency, and proper resource allocation across all active sessions. This section describes the mechanisms and considerations for managing shared context in multi-server MCP deployments over MOQT.

6.1. Context Sharing Architecture

In a shared context scenario, the MCP client acts as a central coordinator that maintains relationships with multiple MCP servers, each providing different capabilities, resources, or specialized services. The client must manage session state, resource subscriptions, and context information across all servers while ensuring proper isolation and security boundaries.

6.1.1. Session Isolation and Coordination

Each MCP server connection maintains its own unique session ID and track namespace to ensure proper isolation:

Server A: mcp/session-a-uuid/resources/documentation
Server B: mcp/session-b-uuid/tools/code_analysis
Server C: mcp/session-c-uuid/prompts/review_templates

The client coordinates these separate sessions while maintaining a unified view of available capabilities across all connected servers.

6.2. Context Synchronization Mechanisms

6.2.1. Resource Context Sharing

When multiple servers provide related resources, the client can implement context synchronization by subscribing to relevant resource tracks from multiple servers and maintaining a unified context state.

6.2.2. Tool Execution Context

Tool executions across multiple servers may require shared context to maintain workflow consistency. Tracking tool execution state across all severs help ensure proper sequencing of multi-server workflows. Similarly results from tools on one server can be automatically passed as parameters to tools on other servers, for example.

6.3. Context Synchronization Benefits

6.3.1. Shared Subscription Management

The client optimizes network usage by implementing intelligent subscription management across multiple servers:

* When multiple servers offer similar resources, the client can subscribe to the most appropriate source based on performance, recency, or quality metrics

* Resource requests can be distributed across servers based on current load, response time, or geographic proximity

* If one server becomes unavailable, the client can seamlessly redirect subscriptions to alternative servers providing similar capabilities

6.3.2. Context Caching and Reuse

The client implements sophisticated caching strategies that work across all connected servers:

* Context information cached from one server can be reused when working with other servers, where appropriate.

* The client tracks resource versions across all servers to ensure cache consistency

* Only changed context elements are synchronized across servers to minimize bandwidth usage

7. Security Considerations

TODO

8. IANA Considerations

TODO

9. Examples

10. References

10.1. Normative References

[MCP] Anthropic, "Model Context Protocol Specification", 18 June 2025, <https://modelcontextprotocol.io/specification/2025-06-18>.

[MOQT] Nandakumar, S., Vasiliev, V., Swett, I., and A. Frindell, "Media over QUIC Transport", 23 June 2025, <https://www.ietf.org/archive/id/draft-ietf-moq-transport-13>.

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

[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/info/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/info/rfc8174>.

10.2. Informative References

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

Appendix A. Acknowledgments

TODO

Authors' Addresses

Cullen Jennings
Cisco Systems
Email: fluffy@cisco.com

Ian Swett
Google
Email: ianswett@google.com

Jonathan Rosenberg
Five9
Email: jdrosen@jdrosen.net

Suhas Nandakumar
Cisco Systems
Email: snandaku@cisco.com