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

S. Nandakumar
Cisco
C. Jennings
Cisco Systems

28 February 2026

MOQ Transport for Agent Protocols
draft-nandakumar-ai-agent-moq-transport-00

Abstract

This document defines a protocol abstraction layer that enables Media over QUIC Transport (MOQT) to serve as a unified transport substrate for inter-agent communication protocols. The abstraction provides a common mapping of request-response and streaming patterns onto MOQT's publish/subscribe model, allowing diverse agent protocols to leverage MOQT's real-time streaming capabilities, built-in prioritization, and efficient multiplexing over QUIC.

The document demonstrates the application of this abstraction to two prominent inter-agent protocols: the Agent-to-Agent (A2A) protocol [A2A], which focuses on agent discovery, task delegation, and collaboration; and the Model Context Protocol (MCP) [MCP], which provides tool and resource access for agents. This unified approach enables interoperability across diverse agent ecosystems while preserving each protocol's semantics.

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://ietf-wg-moq.github.io/draft-a2a-moqt-transport/draft-a2a-moqt-transport.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-nandakumar-ai-agent-moq-transport/.

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/ietf-wg-moq/draft-a2a-moqt-transport.

Status of This Memo

Table of Contents

1. Introduction
  1.1. Requirements Language
2. Transport Requirements and Architecture
3. MOQT Transport Mapping
  3.1. Connection Establishment
  3.2. Track Organization
    3.2.1. Namespace Structure
    3.2.2. Track Categories
  3.3. Message Serialization
  3.4. Priority and Quality of Service
    3.4.1. Group Order
  3.5. Operation Patterns
    3.5.1. Discovery
    3.5.2. Request/Response
    3.5.3. Streaming
4. Protocol Bindings
  4.1. Binding Architecture
  4.2. A2A Binding
    4.2.1. Namespace Structure
    4.2.2. Track Categories
    4.2.3. A2A Operation Mapping
    4.2.4. Message Format
    4.2.5. Priority Mapping
    4.2.6. Streaming Support
    4.2.7. Agent Card Format
  4.3. MCP Binding
  4.4. AutoGen Binding
    4.4.1. Namespace Structure
    4.4.2. Track Categories
    4.4.3. Message Format
    4.4.4. Conversation Patterns
    4.4.5. Priority Mapping
  4.5. Generic JSON-RPC Binding
    4.5.1. Namespace Structure
    4.5.2. Track Categories
    4.5.3. Message Format
    4.5.4. Extension Mechanism
  4.6. Multi-Agent Orchestration Patterns
5. Benefits of MOQT for A2A
6. MOQT Relay Infrastructure for A2A
  6.1. Relay Network Architecture
  6.2. Message Caching Benefits
7. Security Considerations
8. IANA Considerations
9. References
  9.1. Normative References
  9.2. Informative References
Appendix A. Acknowledgments
Authors' Addresses

1. Introduction

The AI agent ecosystem has evolved rapidly, with multiple protocols emerging for different aspects of agent communication:

* *Agent-to-Agent (A2A)* [A2A] focuses on agent discovery, task delegation, and collaboration across platforms

* *Model Context Protocol (MCP)* [MCP] provides tool and resource access for agents

* *Framework-specific protocols* (AutoGen, Semantic Kernel, LangGraph) define multi-agent orchestration patterns

Each of these protocols typically defines its own transport bindings, leading to fragmentation and interoperability challenges across agent ecosystems.

Media over QUIC Transport (MOQT) [MoQ-TRANSPORT] provides a publish/subscribe model over QUIC with hierarchical data organization. MOQT's streaming model, prioritization capabilities, efficient multiplexing, and relay-based architecture make it well-suited as a unified transport substrate for real-time agent communication—enabling scalable message distribution across agent networks.

This document defines a protocol abstraction layer that maps common inter-agent communication patterns—request-response, streaming, and notifications—onto MOQT primitives. The abstraction enables diverse agent protocols to leverage MOQT's capabilities while preserving their native semantics.

The document then demonstrates this abstraction through concrete bindings for two prominent protocols: A2A (which supports JSON-RPC 2.0 over HTTP(S), gRPC, and HTTP+JSON/REST) and MCP. These serve as reference implementations that other inter-agent protocols can follow.

1.1. 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. Transport Requirements and Architecture

According to the A2A specification, all transport protocols MUST provide functional equivalence and support the following capabilities:

* Secure communication over encrypted channels

* Request/response messaging patterns

* Streaming data delivery capabilities

* Error handling and status reporting

* Agent discovery and capability negotiation

* Support for various data types (text, JSON, files)

Figure 1 illustrates how this document layers agent protocols over

the MOQT transport substrate.

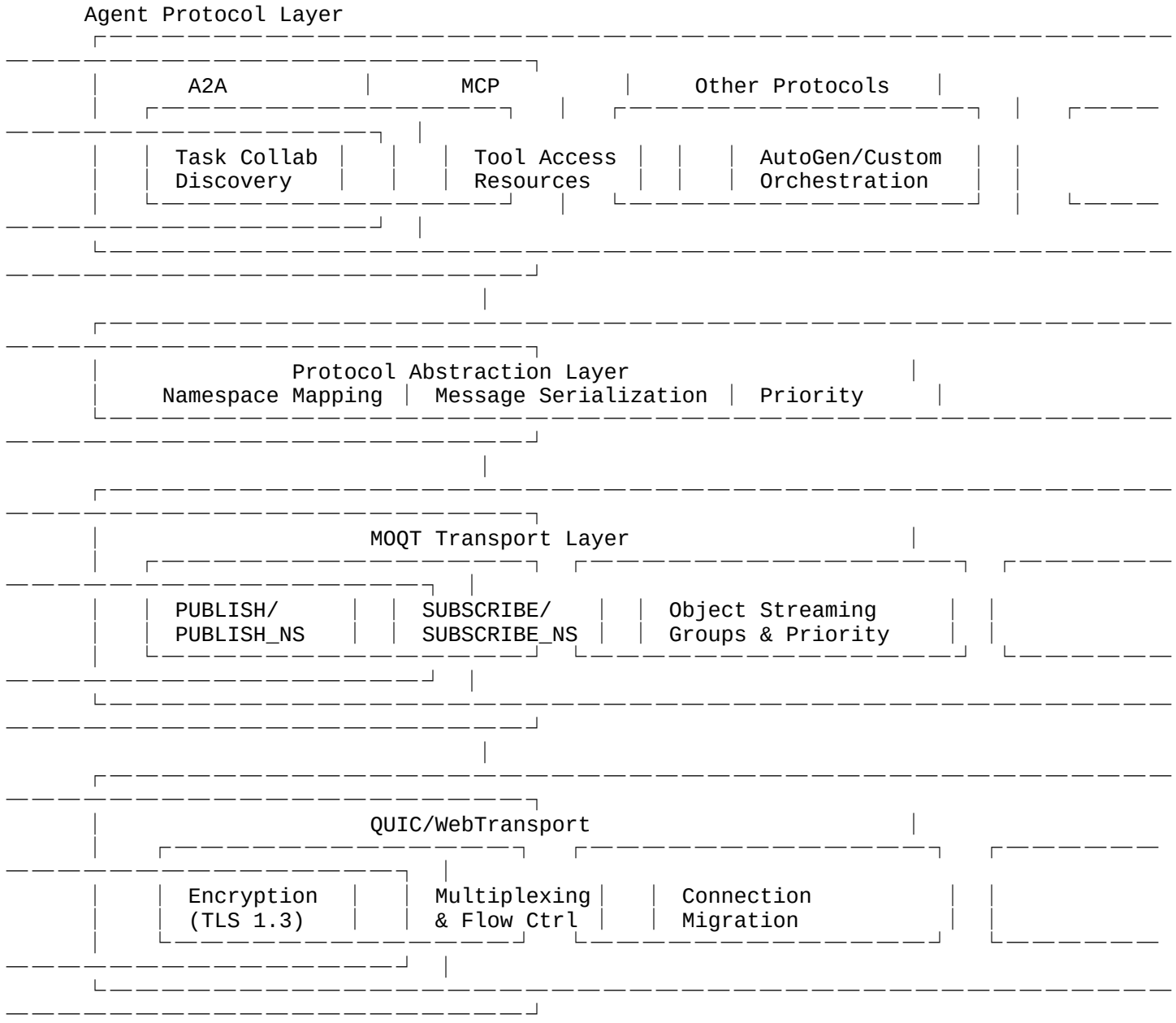
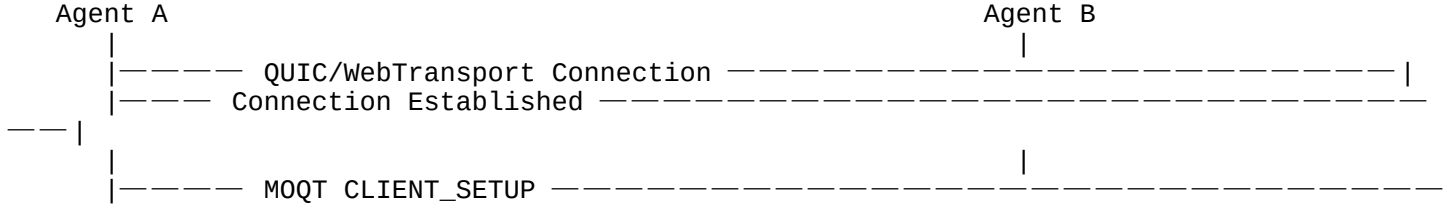


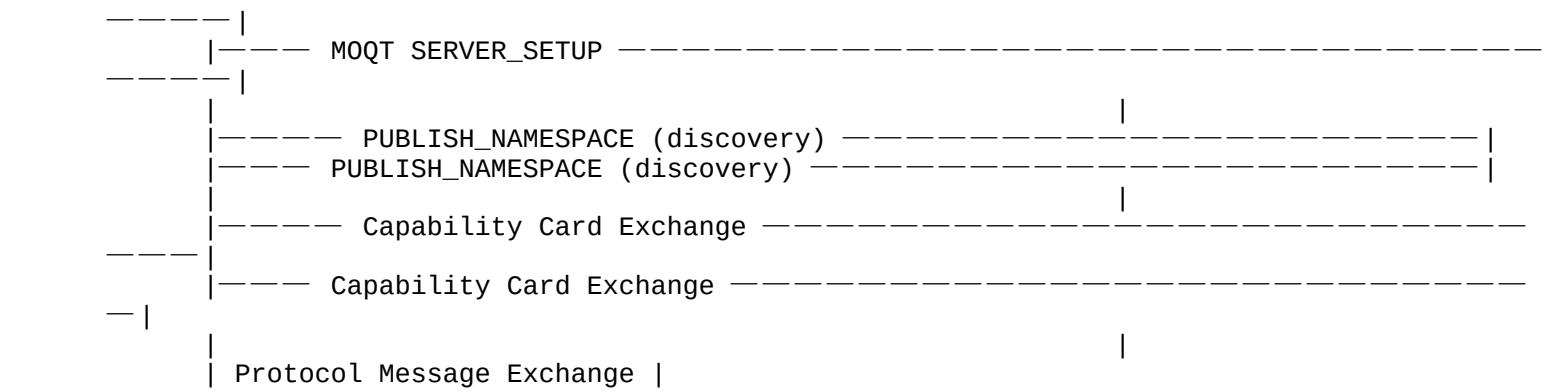
Figure 1: Protocol Layering Architecture

3. MOQT Transport Mapping

3.1. Connection Establishment

Agents using MOQT transport MUST establish either a native QUIC connection or a WebTransport session as defined in [MoQ-TRANSPORT]. For native QUIC connections, agents MUST use the ALPN value "moqt". For WebTransport, the protocol is negotiated using the "WT-Available-Protocols" mechanism as specified in [MoQ-TRANSPORT]. The connection setup follows this sequence:





During the MOQT setup phase, agents MUST negotiate the following setup parameters as MOQT extensions:

* agent-version (0x41475631): Supported agent protocol version string

* agent-protocols (0x41475032): Bitmask of supported agent protocols (A2A=0x01, MCP=0x02)

* agent-auth-schemes (0x41475033): Supported authentication schemes

3.2. Track Organization

Agent protocol messages are mapped to MOQT tracks using a hierarchical namespace structure as defined in [MoQ-TRANSPORT]. MOQT namespaces consist of 1-32 ordered tuple fields, enabling relays to route based on hierarchical prefixes.

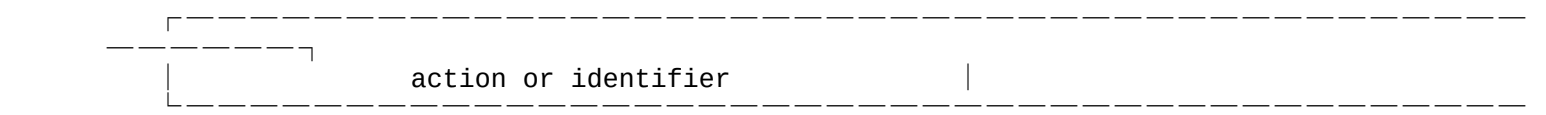
3.2.1. Namespace Structure

Agent protocols over MOQT use a hierarchical namespace structure that enables relay routing and protocol multiplexing. The general pattern is:

Full Track Name Structure:

Track Namespace (tuple fields):

Track Name:



Example Full Track Names:

```
{proto} / session-123 / agent-alice / request -- req-001
{proto} / session-123 / agent-bob / discovery -- agent-card
{proto} / session-123 / agent-alice / stream -- task-456
```

The first namespace field identifies the protocol binding (e.g., "a2a", "mcp", "autogen") enabling multiple protocols to coexist on shared relay infrastructure. Protocol-specific namespace structures are defined in their respective binding sections (Section 4.2, Section 4.3).

3.2.2. Track Categories

Agent protocols typically organize communication into the following logical track categories. Protocol bindings define specific namespace patterns for each category.

Request Tracks: Used for outbound request messages. Track name typically contains a request identifier for correlation.

Response Tracks: Used for response messages. Track name contains the correlated request identifier.

Resource/State Tracks: Used for resource access or state synchronization. Track name identifies the specific resource or state key.

Notification Tracks: Used for push notifications and asynchronous events.

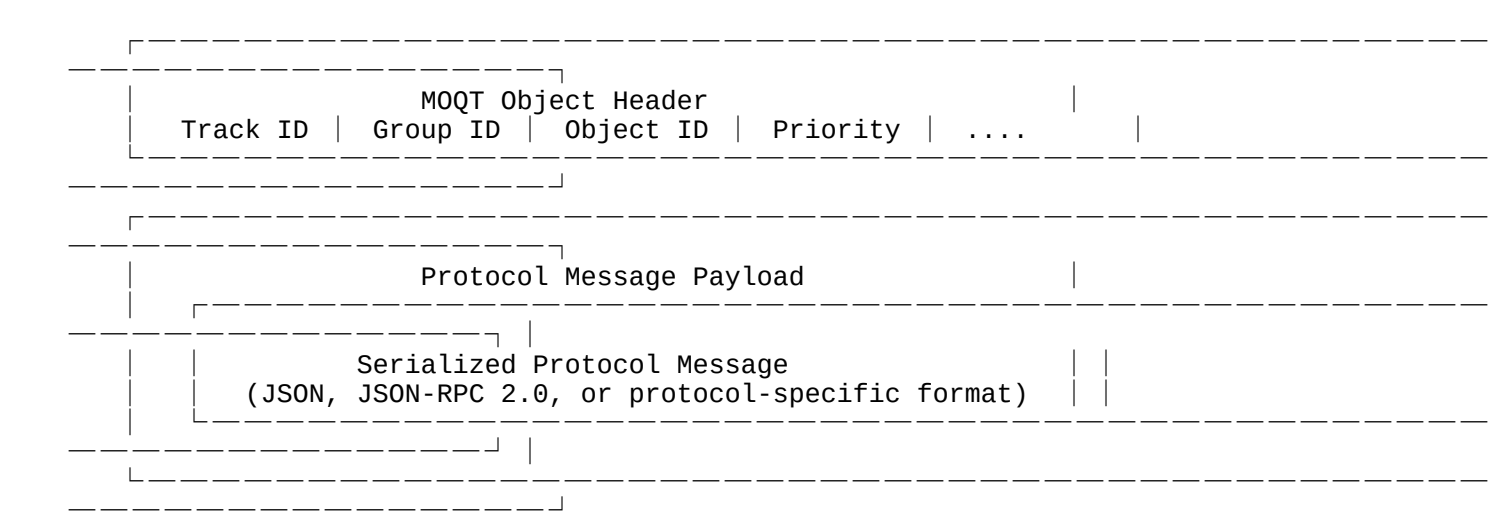
Discovery Tracks: Used for agent/service discovery and capability exchange.

Protocol-specific track categories and namespace patterns are defined in their respective binding sections.

3.3. Message Serialization

Agent protocol messages are serialized and encapsulated within MOQT objects. The serialization format is protocol-specific, but most agent protocols use JSON or JSON-RPC 2.0. Each MOQT object contains:

MOQT Object Structure:



Protocol bindings specify the exact serialization format and any required message envelope structure. Bindings SHOULD preserve native message formats where possible to maintain compatibility with existing protocol implementations.

3.4. Priority and Quality of Service

MOQT uses a 0-255 priority scale where lower values indicate higher priority. Protocol bindings map their message types to MOQT Publisher Priority ranges. The following general priority tiers are recommended:

Message Category	MOQT Priority	Description

Critical/Cancel	0-31	Critical errors, cancellations, urgent control
Interactive Response	32-63	Responses requiring immediate delivery
Standard Request	64-95	Normal request/response messages
Streaming Updates	96-127	Incremental status and data updates
Discovery	128-159	Service/agent discovery and capability exchange
Background	160-191	Non-critical notifications, telemetry
Bulk Transfer	192-255	Large file transfers, batch operations

Table 1

Protocol bindings SHOULD define specific mappings from their message types to these priority ranges. See Section 4.2 and Section 4.3 for protocol-specific priority assignments.

3.4.1. Group Order

For streaming operations, MOQT Group Order determines whether groups are delivered in ascending (oldest first) or descending (newest first) order:

* Sequential processing streams: Ascending order (preserve execution sequence)

* Live notifications: Descending order (prioritize recent events)

* Ordered data transfers: Ascending order (sequential chunk delivery)

3.5. Operation Patterns

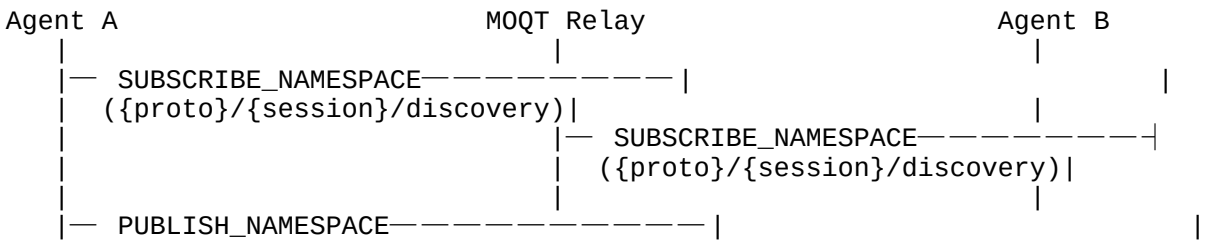
This section defines common operation patterns that protocol bindings use to implement agent communication over MOQT.

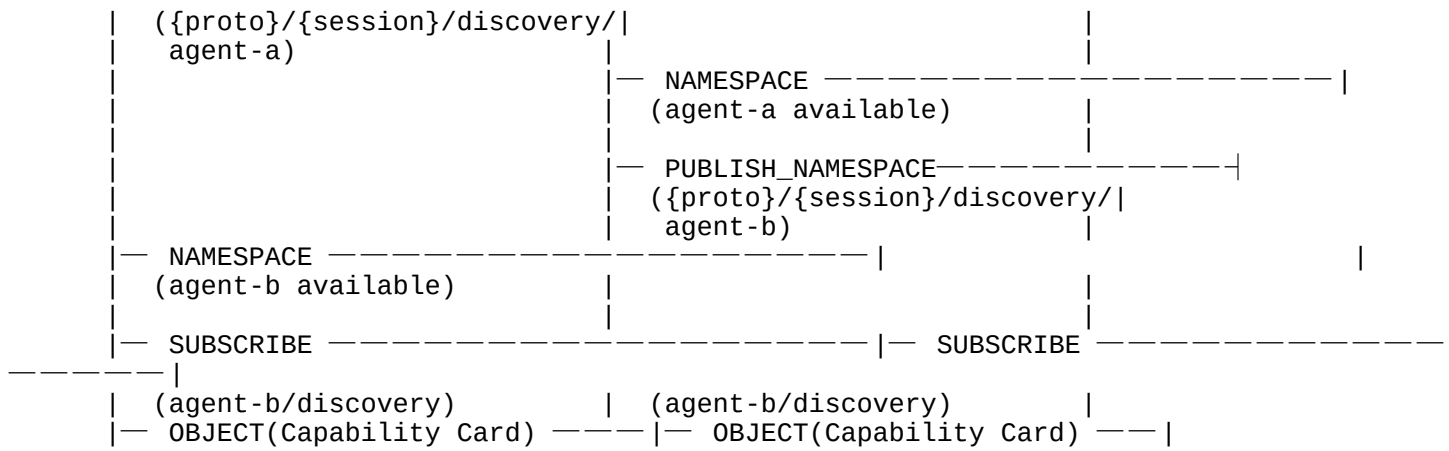
3.5.1. Discovery

Agent discovery over MOQT leverages the SUBSCRIBE_NAMESPACE and PUBLISH_NAMESPACE mechanisms defined in [MoQ-TRANSPORT] for efficient in-band discovery of agents within a session context.

Agents use PUBLISH_NAMESPACE to advertise their presence and SUBSCRIBE_NAMESPACE to discover other agents:

Discovery Flow Using Namespace Operations:





Each agent publishes a capability card containing its identity, capabilities, and supported operations. The card serves as the discovery mechanism through which agents advertise their presence and abilities to potential collaborators. The card is published as an MOQT object on the agent's discovery track.

A capability card typically includes:

* *Identity*: Name, description, and endpoint URL

* *Capabilities*: Supported features such as streaming or notifications

* *Authentication*: Supported authentication schemes

* *MOQT Extension*: Transport-specific configuration for MOQT connectivity

Protocol-specific card formats are defined in their respective binding sections (e.g., A2A Agent Card in Section 4.2.7).

3.5.2. Request/Response

Request/response interactions between agents are implemented using coordinated PUBLISH_NAMESPACE, SUBSCRIBE_NAMESPACE, and object delivery. This pattern enables agents to exchange messages through MOQT relays without requiring direct connectivity.

The following table describes how common agent communication patterns map to MOQT primitives:

Operation Pattern	MOQT Mechanism	Description
One-shot request	PUBLISH + OBJECT	Single request published to a track
Streaming request	PUBLISH + multiple OBJECTs	Request with incremental data across objects
Resource fetch	SUBSCRIBE + OBJECT	Subscribe to retrieve a specific resource
Resource listing	SUBSCRIBE_NAMESPACE	Discover available resources via namespace
Cancellation	OBJECT on control track	Cancel signal published to control track

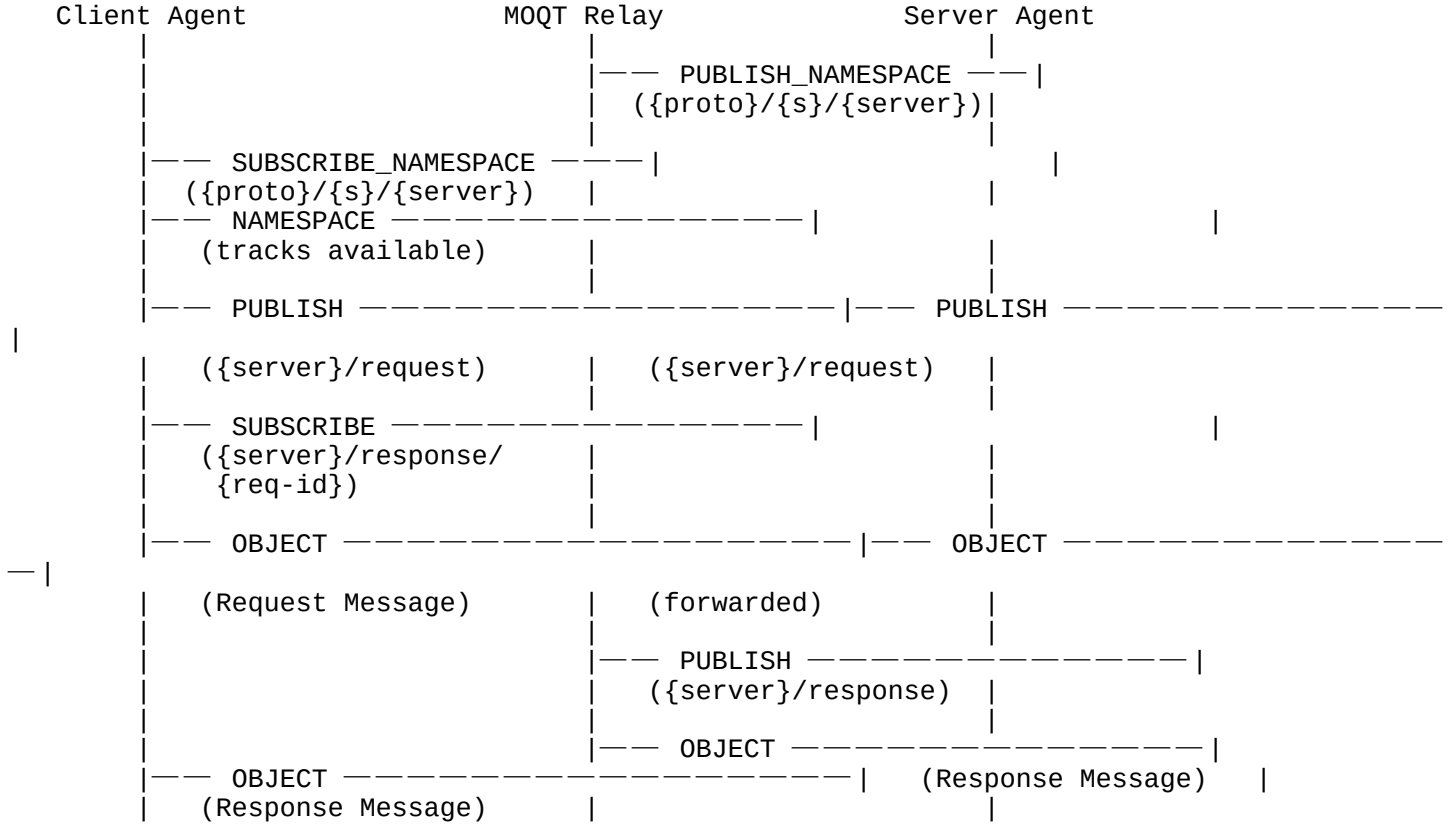
Subscription	SUBSCRIBE (ongoing)	Long-lived subscription for updates

Table 2

Protocol-specific operation mappings are defined in their respective binding sections.

The following diagram illustrates the message flow for a typical request/response interaction between a client agent and a server agent through an MOQT relay. The server agent first publishes its namespace to advertise availability. The client discovers the server via namespace subscription, then publishes a request and subscribes to the corresponding response track. The server processes the request and publishes the response, which the relay forwards to the subscribed client.

Request/Response Flow:



Request-response correlation uses message IDs embedded in the protocol payload (e.g., JSON-RPC id field). Both request and response track names include the request-specific ID to enable efficient subscription filtering and correlation.

3.5.3. Streaming

Long-running operations like task execution or file transfer utilize MOQT's streaming capabilities. Stream tracks carry incremental updates as separate MOQT objects organized into groups.

Streaming leverages MOQT's object model to deliver incremental updates. The client subscribes to an operation-specific track and receives a sequence of events as the server executes the operation. Each event is published as an MOQT object, organized into groups that represent logical phases of execution.

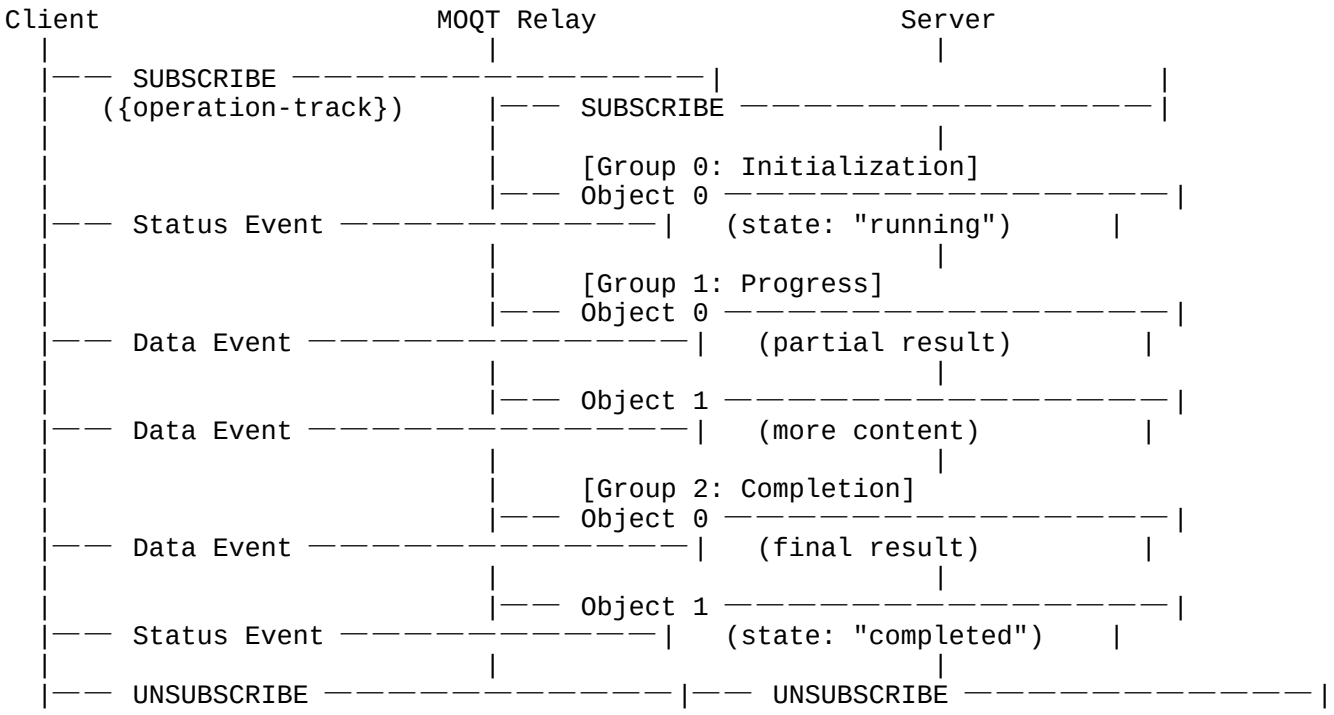
Protocol bindings define specific event types for their streaming operations. Common patterns include:

Status Events: Published when operation state changes. Contains operation ID, current state, and optional status message.

Data Events: Published when operation produces output. Contains result data with incremental or complete delivery.

The following diagram illustrates a generic streaming flow. The client subscribes to the operation track. As the server executes, it publishes status and data updates as MOQT objects. Objects are organized into groups representing execution phases. The relay forwards each object to the subscribed client in real-time.

Streaming Operation Flow:



Groups serve as synchronization points within the stream. The server advances to a new group when transitioning between execution phases, allowing clients to identify and process related updates together.

3.5.3.1. Late Joining and Caching

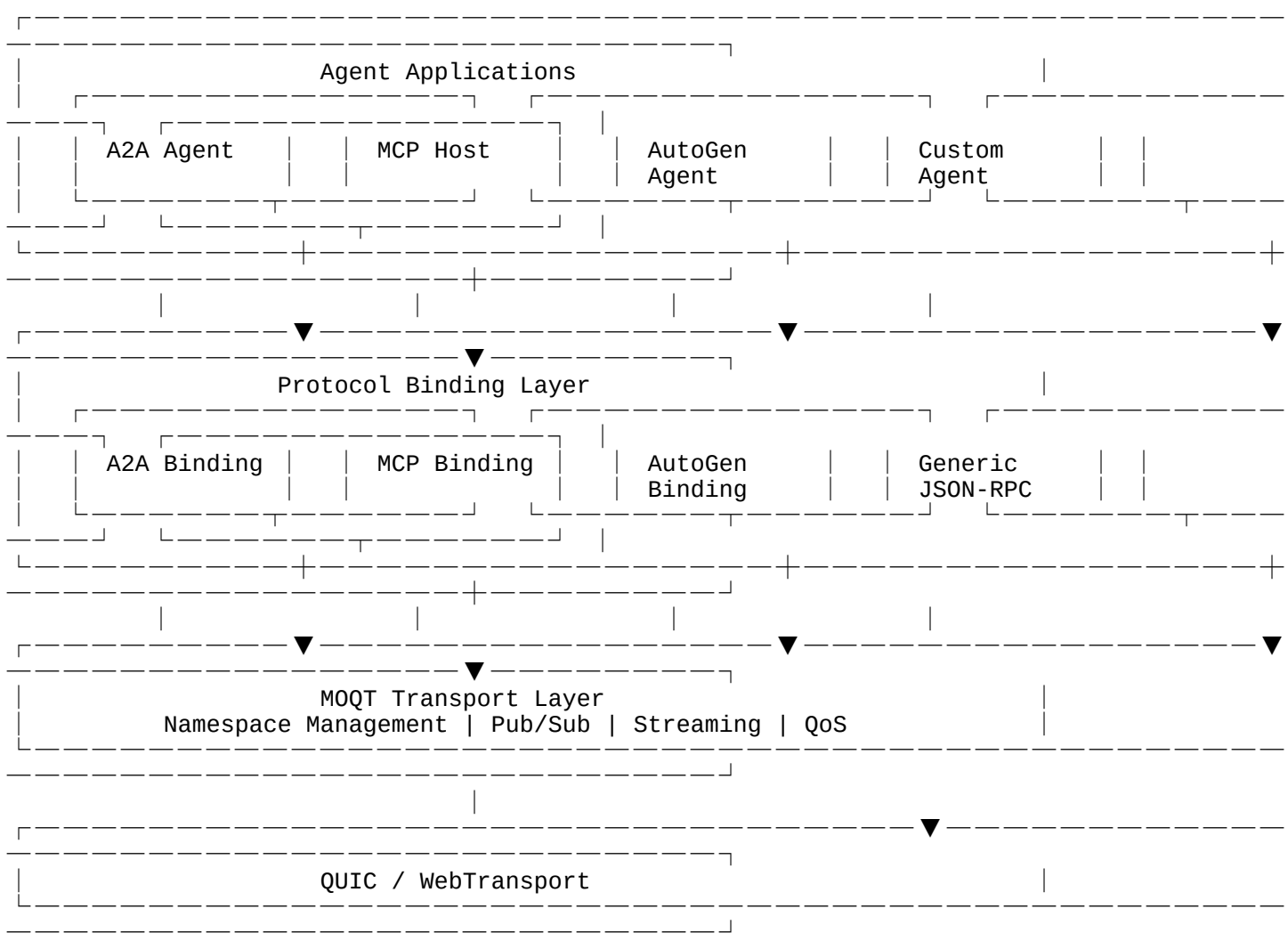
MOQT groups serve as join points for late-arriving subscribers. Relays MAY cache recent groups to enable:

* Clients joining mid-stream receive the latest group
* Recovery from transient disconnections
* Efficient replay of recent operation history

4. Protocol Bindings

To enable MOQT as a unified transport for multiple agent communication protocols, this document defines protocol bindings that map protocol-specific semantics to the transport patterns defined in Section 3.

4.1. Binding Architecture



Each protocol binding provides the following elements for MOQT integration:

* *Protocol Identifier*: A unique string identifying the protocol (e.g., "a2a", "mcp", "autogen") used to distinguish bindings.

* *Version*: The protocol version supported by the binding, enabling version negotiation and compatibility detection.

* *Namespace Prefix*: A tuple of strings forming the root namespace for the protocol's tracks, enabling namespace partitioning across different protocols on shared infrastructure.

* *Message Serialization*: Methods to convert protocol messages to bytes for MOQT object payloads and to parse received bytes back into protocol messages.

* *Track Mapping*: Logic to map protocol operations to appropriate MOQT track names, determining which track should carry each message type.

* *Priority Mapping*: Rules to assign MOQT publisher priorities based on message types, ensuring appropriate quality of service for different operations.

* *Namespace Event Handling*: Callbacks for processing MOQT namespace events such as ANNOUNCE and SUBSCRIBE_NAMESPACE, enabling protocol- specific discovery and subscription management.

Different protocols use distinct namespace prefixes to enable coexistence on shared MOQT infrastructure:

Protocol	Namespace Prefix	Example Namespace	Example Track Name
A2A	a2a/{session}/{agent}	a2a/s1/agent-a/ task	t1
MCP	mcp/{session}/{server}	mcp/s1/fs-server/tools	read
AutoGen	autogen/{session}/{runtime}	autogen/s1/rt1/ agent	msg
Generic	agent/{session}/{id}	agent/s1/custom/ rpc	call

Table 3

4.2. A2A Binding

The Agent-to-Agent Protocol [A2A] binding maps A2A's task delegation and collaboration semantics to MOQT primitives using the patterns defined in Section 3.

4.2.1. Namespace Structure

A2A uses the a2a namespace prefix with the following hierarchy:

a2a/{session-id}/{agent-id}/{category}

Where:

* session-id: Unique identifier for the collaboration session
* agent-id: Unique identifier for the publishing agent
* category: Message category (request, response, task, notify, discovery)

4.2.2. Track Categories

Category	Purpose	Track Name Pattern
request	Outbound requests	{request-id}
response	Request responses	{request-id}
task	Task lifecycle	{task-id}
notify	Push notifications	{notification-type}
discovery	Agent cards	agent-card

Table 4

4.2.3. A2A Operation Mapping

The following table maps A2A v0.3.0 operations to MOQT primitives. The Generic Pattern column references patterns from Section 3.5.2.

A2A Operation	MOQT Mechanism	Track Pattern	Generic Pattern
SendMessage	PUBLISH + OBJECT	{agent}/request -- {req-id}	One-shot request
SendStreamingMessage	PUBLISH + multiple OBJECTs	{agent}/stream -- {req-id}	Streaming request
GetTask	SUBSCRIBE + OBJECT fetch	{agent}/task -- {task-id}	Resource fetch
ListTasks	SUBSCRIBE_NAMESPACE	{agent}/task	Resource listing
CancelTask	OBJECT on request track	{agent}/request -- cancel-{task-id}	Cancellation
SubscribeToTask	SUBSCRIBE (ongoing)	{agent}/task -- {task-id}	Subscription
GetExtendedAgentCard	SUBSCRIBE + OBJECT	{agent}/discovery -- agent-card	Resource fetch

Table 5

4.2.4. Message Format

A2A messages are serialized as JSON-RPC 2.0 payloads within MOQT objects. The binding preserves A2A's native message format without transformation.

4.2.5. Priority Mapping

A2A Message Type	MOQT Priority Range
Cancel/Error	0-31
SendMessage response	32-63
SendMessage request	64-95
Task updates	96-127
Discovery	128-159
Background	160-255

Table 6

4.2.6. Streaming Support

A2A streaming operations (SendStreamingMessage, task subscriptions) map to MOQT groups and objects following the pattern in Section 3.5.3.

A2A defines two streaming event types:

TaskStatusUpdateEvent: Published when task state changes. Contains task ID, current state (queued, running, completed, failed,

canceled), and optional message.

TaskArtifactUpdateEvent: Published when task produces output. Contains artifact data (text, files, structured data) with incremental or complete delivery.

The mapping to MOQT objects:

* Each streaming response creates a new MOQT group

* Incremental updates are individual objects within the group

* TaskStatusUpdateEvent and TaskArtifactUpdateEvent map to objects

* Groups represent execution phases (initialization, progress, completion)

4.2.7. Agent Card Format

Per A2A v0.3.0, each agent publishes an Agent Card containing identity, capabilities, and supported operations. The A2A Agent Card extends the generic agent card structure with A2A-specific fields:

```
{
  "name": "example-agent",
  "description": "An example A2A agent",
  "url": "https://example.com/agent",
  "protocolVersion": "0.3.0",
  "capabilities": {
    "streaming": true,
    "pushNotifications": true
  },
  "skills": [],
  "authentication": {
    "schemes": ["bearer", "oauth2"]
  },
  "moqt": {
    "namespace": "a2a/session-123/agent-example",
    "relayEndpoint": "moqt://relay.example.com:4443",
    "supportedExtensions": ["a2a-version", "a2a-protocols"]
  },
  "signature": "base64-encoded-signature"
}
```

The A2A-specific fields include:

* protocolVersion: The A2A protocol version supported (e.g., "0.3.0")

* capabilities: A2A capability flags including streaming and pushNotifications

* skills: Array of skill definitions the agent can perform

* signature: Optional cryptographic signature for card verification

The moqt extension provides MOQT transport configuration with the A2A namespace prefix.

4.3. MCP Binding

The Model Context Protocol [MCP] can be transported over MOQT to enable tool and resource access for agents. The complete specification for MCP over MOQT is defined in [MCP-MOQT].

[MCP-MOQT] defines dedicated MOQT tracks for each MCP primitive:

control tracks for session management and capability negotiation, resource tracks for server-published content delivery, tool tracks for invocation requests and responses, prompt tracks for template distribution, and notification tracks for asynchronous events (Section 3). The specification covers protocol operations including session establishment, priority management for ensuring critical operations receive sufficient bandwidth, and comprehensive error handling (Section 4).

A key contribution of [MCP-MOQT] is the Agent Skills architecture (Section 6), which extends AI capabilities beyond atomic tool operations. Skills provide composed instructions for complex tasks and use progressive loading (metadata, instructions, resources) that aligns naturally with MOQT's object-based delivery, enabling efficient bandwidth utilization and aggressive caching. The specification also describes relay support (Section 5) for scalable MCP deployments, including subscription aggregation where multiple client subscriptions are consolidated into single upstream requests, and content caching strategies optimized for AI workflows.

4.4. AutoGen Binding

AutoGen [AutoGen] is a multi-agent conversation framework that enables complex LLM applications through agent collaboration. This binding defines how AutoGen's conversation patterns map to MOQT.

4.4.1. Namespace Structure

AutoGen uses the autogen namespace prefix:

autogen/{session-id}/{runtime-id}/{agent-name}

Where:

* session-id: Unique identifier for the AutoGen session

* runtime-id: Identifier for the AutoGen runtime instance

* agent-name: Name of the AutoGen agent

4.4.2. Track Categories

Category	Purpose	Track Name Pattern
message	Inter-agent messages	{conversation-id}
control	Runtime control	{command}
state	Agent state sync	{state-key}
result	Final outputs	{task-id}

Table 7

4.4.3. Message Format

AutoGen messages are serialized as JSON with the following structure:

```
{
  "sender": "agent-name",
  "recipient": "agent-name | broadcast",
  "content": "message content or structured data",
  "metadata": {
    "conversation_id": "conv-123",
```

```
      "turn": 5,
      "role": "assistant | user | system"
    }
  }
```

4.4.4. Conversation Patterns

AutoGen's conversation patterns map to MOQT as follows:

Two-Agent Chat: Each agent publishes to their message track. Agents subscribe to their conversation partner's track.

Group Chat: A coordinator agent manages turn-taking. All agents subscribe to the coordinator's broadcast track.

Nested Chat: Hierarchical namespaces enable nested conversations: autogen/{session}/runtime-1/outer-agent/inner-runtime/inner-agent

4.4.5. Priority Mapping

AutoGen Message Type	MOQT Priority Range
Termination signals	0-31
Human input requests	32-63
Agent responses	64-95
Function results	96-127
State updates	128-159
Logging/debug	160-255

Table 8

4.5. Generic JSON-RPC Binding

This binding provides a minimal baseline for custom agent protocols that use JSON-RPC 2.0 messaging. It serves as an extensible foundation for protocols not covered by specific bindings.

4.5.1. Namespace Structure

Generic protocols use the agent namespace prefix:

agent/{session-id}/{agent-id}/{protocol-name}

Where:

* session-id: Unique session identifier

* agent-id: Unique agent identifier

* protocol-name: Custom protocol identifier

4.5.2. Track Categories

Category	Purpose	Track Name Pattern
rpc	Request/response	{request-id}
stream	Streaming data	{stream-id}

event	Async events	{event-type}

Table 9

4.5.3. Message Format

Messages follow standard JSON-RPC 2.0:

```
{
  "jsonrpc": "2.0",
  "method": "protocol.method",
  "params": {},
  "id": "request-id"
}
```

4.5.4. Extension Mechanism

Custom protocols can extend this binding by:

1. Defining additional track categories
2. Specifying custom message schemas within the JSON-RPC params
3. Registering protocol-specific priorities
4. Adding custom metadata in MOQT extension headers

Implementations SHOULD document their extensions and MAY register them in the IANA Protocol Bindings Registry.

4.6. Multi-Agent Orchestration Patterns

The abstraction layer supports common multi-agent coordination patterns:

TODO: Discuss each of the following patterns:

* Sequential Orchestration
* Concurrent (Broadcast) Pattern
* Handoff Pattern
* Hierarchical Teams

5. Benefits of MOQT for A2A

MOQT provides significant advantages over traditional A2A transports, particularly for real-time agent collaboration and large-scale deployments.

Feature	HTTP/JSON-RPC	gRPC	REST	MOQT
Pub/Sub Native	No	Limited	No	Yes
Multi-party Broadcast	Polling	N/A	Polling	Native
Relay/CDN Support	Separate infra	N/A	CDN possible	Built-in

Priority QoS	Application layer	Per-stream	None	Per-object (0-255)
Connection Migration	No	Limited	No	QUIC native
Late Join Support	N/A	N/A	N/A	Group-based
Streaming Granularity	Message	Stream	N/A	Object/Group
Discovery Mechanism	Well-known URI	Service registry	Well-known URI	SUBSCRIBE_NAMESPACE

Table 10

MOQT enables real-time agent communication through low-latency streaming, priority-based message delivery, and built-in flow control that prevents overload during high-volume interactions.

The publish/subscribe model provides scalability through one-to-many broadcasts, dynamic in-band discovery via SUBSCRIBE_NAMESPACE, and relay-based distribution that prevents connection explosion in large mesh networks.

MOQT delivers reliability and resilience via QUIC connection migration, automatic retry mechanisms, graceful degradation, and object caching at relays for recovery from transient disconnections.

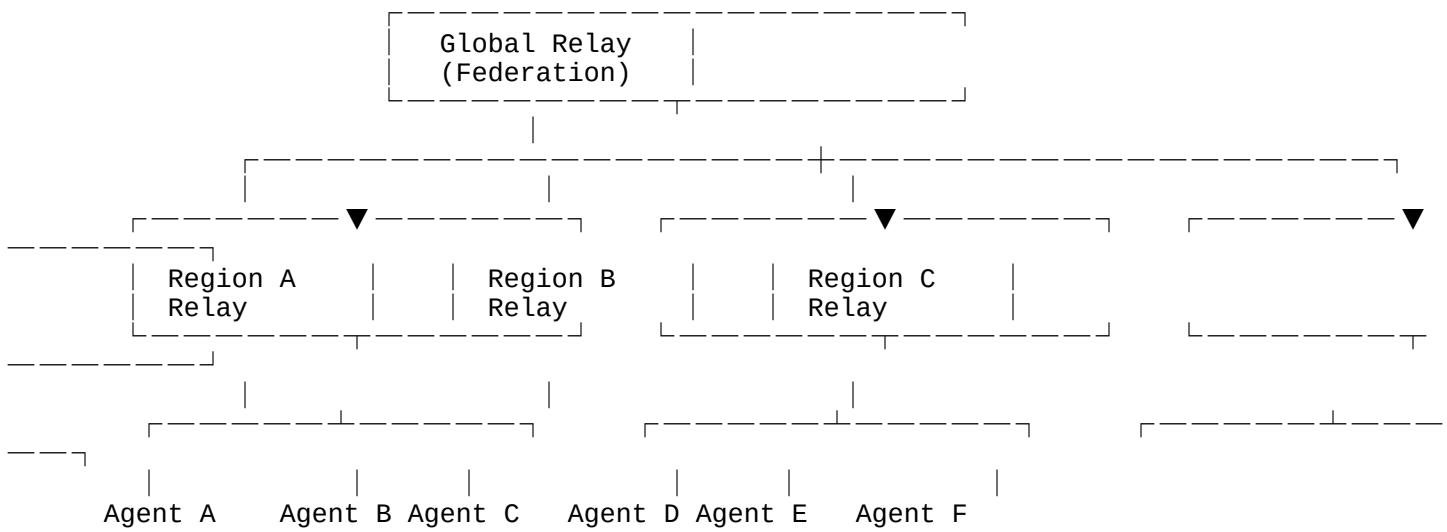
6. MOQT Relay Infrastructure for A2A

MOQT relays provide critical infrastructure benefits for large-scale A2A deployments, enabling efficient message distribution and caching.

6.1. Relay Network Architecture

MOQT relays form a hierarchical distribution network that optimizes A2A message delivery across geographic and organizational boundaries:

Hierarchical Relay Network:



This hierarchical structure provides:

* Regional optimization with local agent clusters

* Global connectivity for cross-region agent collaboration

* Load distribution to prevent single points of failure

* Configurable routing policies based on agent requirements

6.2. Message Caching Benefits

MOQT relays implement intelligent caching strategies optimized for A2A communication patterns. Caching tiers range from hot caches with short TTLs for real-time responses and task status updates, to warm caches for agent capability profiles and session metadata, to cold storage for historical task logs and compliance data.

7. Security Considerations

TODO

8. IANA Considerations

TODO

9. References

9.1. Normative References

[A2A] A2A Project, "Agent-to-Agent Protocol Specification", Version 0.3.0, 2025, <https://google.github.io/A2A/specification/>.

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

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[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

[AutoGen] Wu, Q., Banber, G., Zhang, Y., Wu, Y., Li, B., Zhu, E., and A. Awadallah, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation", arXiv 2308.08155, 2023, <https://arxiv.org/abs/2308.08155>.

[MCP-MOQT] Jennings, C., Swett, I., Rosenberg, J., and S. Nandakumar, "Model Context Protocol and Agent Skills over Media over QUIC Transport", Work in Progress, Internet-Draft, draft-mcp-over-moqt, 2025, <https://datatracker.ietf.org/doc/draft-mcp-over-moqt/>.

[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>.

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

Appendix A. Acknowledgments

TODO

Authors' Addresses

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com

Cullen Jennings
Cisco Systems
Email: fluffy@cisco.com